

LD A,(60H)

0	0	1	1	1	0	1	0
0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0

LD B,A

0	1	0	0	0	1	1	1
---	---	---	---	---	---	---	---

ADD A,B

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

A	S	S	E	M	B	L	Y
L	A	N	G	U	A	G	E
T U T O R							



DISK

TRS-80 MODEL I/III



CAT. NO. 26-2018

TERMS AND CONDITIONS OF SALE AND LICENSE OF RADIO SHACK COMPUTER EQUIPMENT AND SOFTWARE
PURCHASED FROM A RADIO SHACK COMPANY-OWNED COMPUTER CENTER, RETAIL STORE OR FROM A
RADIO SHACK FRANCHISEE OR DEALER AT ITS AUTHORIZED LOCATION

LIMITED WARRANTY

I. CUSTOMER OBLIGATIONS

- A. CUSTOMER assumes full responsibility that this Radio Shack computer hardware purchased (the "Equipment"), and any copies of Radio Shack software included with the Equipment or licensed separately (the "Software") meets the specifications, capacity, capabilities, versatility, and other requirements of CUSTOMER.
- B. CUSTOMER assumes full responsibility for the condition and effectiveness of the operating environment in which the Equipment and Software are to function, and for its installation.

II. RADIO SHACK LIMITED WARRANTIES AND CONDITIONS OF SALE

- A. For a period of ninety (90) calendar days from the date of the Radio Shack sales document received upon purchase of the Equipment, RADIO SHACK warrants to the original CUSTOMER that the Equipment and the medium upon which the Software is stored is free from manufacturing defects. THIS WARRANTY IS ONLY APPLICABLE TO PURCHASES OF RADIO SHACK EQUIPMENT BY THE ORIGINAL CUSTOMER FROM RADIO SHACK COMPANY-OWNED COMPUTER CENTERS, RETAIL STORES AND FROM RADIO SHACK FRANCHISEES AND DEALERS AT ITS AUTHORIZED LOCATION. The warranty is void if the Equipment's case or cabinet has been opened, or if the Equipment or Software has been subjected to improper or abnormal use. If a manufacturing defect is discovered during the stated warranty period, the defective Equipment must be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer for repair, along with a copy of the sales document or lease agreement. The original CUSTOMER'S sole and exclusive remedy in the event of a defect is limited to the correction of the defect by repair, replacement, or refund of the purchase price, at RADIO SHACK'S election and sole expense. RADIO SHACK has no obligation to replace or repair expendable items.
- B. RADIO SHACK makes no warranty as to the design, capability, capacity, or suitability for use of the Software, except as provided in this paragraph. Software is licensed on an "AS IS" basis, without warranty. The original CUSTOMER'S exclusive remedy, in the event of a Software manufacturing defect, is its repair or replacement within thirty (30) calendar days of the date of the Radio Shack sales document received upon license of the Software. The defective Software shall be returned to a Radio Shack Computer Center, a Radio Shack retail store, participating Radio Shack franchisee or Radio Shack dealer along with the sales document.
- C. Except as provided herein no employee, agent, franchisee, dealer or other person is authorized to give any warranties of any nature on behalf of RADIO SHACK.
- D. Except as provided herein, **RADIO SHACK MAKES NO WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**
- E. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation(s) may not apply to CUSTOMER.

III. LIMITATION OF LIABILITY

- A. EXCEPT AS PROVIDED HEREIN, RADIO SHACK SHALL HAVE NO LIABILITY OR RESPONSIBILITY TO CUSTOMER OR ANY OTHER PERSON OR ENTITY WITH RESPECT TO ANY LIABILITY, LOSS OR DAMAGE CAUSED OR ALLEGED TO BE CAUSED DIRECTLY OR INDIRECTLY BY "EQUIPMENT" OR "SOFTWARE" SOLD, LEASED, LICENSED OR FURNISHED BY RADIO SHACK, INCLUDING, BUT NOT LIMITED TO, ANY INTERRUPTION OF SERVICE, LOSS OF BUSINESS OR ANTICIPATORY PROFITS OR CONSEQUENTIAL DAMAGES RESULTING FROM THE USE OR OPERATION OF THE "EQUIPMENT" OR "SOFTWARE". IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS, OR ANY INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR IN ANY MANNER ARISING OUT OF OR CONNECTED WITH THE SALE, LEASE, LICENSE, USE OR ANTICIPATED USE OF THE "EQUIPMENT" OR "SOFTWARE".
NOTWITHSTANDING THE ABOVE LIMITATIONS AND WARRANTIES, RADIO SHACK'S LIABILITY HEREUNDER FOR DAMAGES INCURRED BY CUSTOMER OR OTHERS SHALL NOT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE PARTICULAR "EQUIPMENT" OR "SOFTWARE" INVOLVED.
- B. RADIO SHACK shall not be liable for any damages caused by delay in delivering or furnishing Equipment and/or Software.
- C. No action arising out of any claimed breach of this Warranty or transactions under this Warranty may be brought more than two (2) years after the cause of action has accrued or more than four (4) years after the date of the Radio Shack sales document for the Equipment or Software, whichever first occurs.
- D. Some states do not allow the limitation or exclusion of incidental or consequential damages, so the above limitation(s) or exclusion(s) may not apply to CUSTOMER.

IV. RADIO SHACK SOFTWARE LICENSE

RADIO SHACK grants to CUSTOMER a non-exclusive, paid-up license to use the RADIO SHACK Software on **one** computer, subject to the following provisions:

- A. Except as otherwise provided in this Software License, applicable copyright laws shall apply to the Software.
- B. Title to the medium on which the Software is recorded (cassette and/or diskette) or stored (ROM) is transferred to CUSTOMER, but not title to the Software.
- C. CUSTOMER may use Software on one host computer and access that Software through one or more terminals if the Software permits this function.
- D. CUSTOMER shall not use, make, manufacture, or reproduce copies of Software except for use on **one** computer and as is specifically provided in this Software License. Customer is expressly prohibited from disassembling the Software.
- E. CUSTOMER is permitted to make additional copies of the Software **only** for backup or archival purposes or if additional copies are required in the operation of **one** computer with the Software, but only to the extent the Software allows a backup copy to be made. However, for TRSDOS Software, CUSTOMER is permitted to make a limited number of additional copies for CUSTOMER'S own use.
- F. CUSTOMER may resell or distribute unmodified copies of the Software provided CUSTOMER has purchased one copy of the Software for each one sold or distributed. The provisions of this Software License shall also be applicable to third parties receiving copies of the Software from CUSTOMER.
- G. All copyright notices shall be retained on all copies of the Software.

V. APPLICABILITY OF WARRANTY

- A. The terms and conditions of this Warranty are applicable as between RADIO SHACK and CUSTOMER to either a sale of the Equipment and/or Software License to CUSTOMER or to a transaction whereby RADIO SHACK sells or conveys such Equipment to a third party for lease to CUSTOMER.
- B. The limitations of liability and Warranty provisions herein shall inure to the benefit of RADIO SHACK, the author, owner and/or licensor of the Software and any manufacturer of the Equipment sold by RADIO SHACK.

VI. STATE LAW RIGHTS

The warranties granted herein give the **original** CUSTOMER specific legal rights, and the **original** CUSTOMER may have other rights which vary from state to state.



Assembly-Language Tutor

(Disk Version)

by

William Barden, Jr.

A self-teaching guide for assembly-language programming on the
Radio Shack TRS-80 Model I and Model III Computers

Note

This manual and the related software are special and are for educational, self-teaching purposes only. They are not intended for use as general program development tools. The Assembler and Interpreter contain built-in checks designed to aid the assembly-language beginner; these checks may inhibit implementation of larger or sophisticated assembly-language programs. Use this package to learn assembly language; use the Series I Editor/Assembler as a full-fledged Model I or III Editor/Assembler.

If you have questions about this package please contact your local Radio Shack store.

©1983 by Radio Shack, a division of Tandy
Corporation, Fort Worth, Texas 76102

FIRST EDITION
FIRST PRINTING

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number:
Library of Congress Catalog Card Number:

Printed in USA

Preface

The Assembly-Language Tutor for TRS-80 Model I and III Assembly-Language Programming is a self-contained course that will teach you the rudiments of assembly-language programming on the Model I or Model III Computers.

Assembly-language programming allows you to execute “Z-80” machine-language instructions on the Model I or III. These are instructions that operate directly with the Z-80 microprocessor used in the Models I and III.

Assembly language is a great deal faster than BASIC or other “high-level” languages for the Model I and III — as much as 300 times faster, depending upon the task. On the other hand, assembly language is much harder to learn than BASIC.

What we’ve attempted to do here is to make assembly language as “foolproof” as possible, with an “interactive” package to guide you through the maze of assembly-language instructions and techniques.

The Assembly-Language Tutor includes not only a complete text on beginning assembly-language programming, but a complete software training package as well.

The software portion of the Assembly-Language Tutor includes a complete editor, assembler (to translate text into Z-80 machine-language instructions), and control program.

The control program allows you to execute assembly-language programs, and has a great deal of error checking built into it. These checks will not permit you to “lose control” as is the case with just an assembler. The control program and its main component, an “interpreter,” constantly monitor the assembly-language instruction you are executing. They will inform you if you have made a serious error that in other situations would “crash” the system.

At any given time you can see the state of the assembly-language program displayed on the screen (text and instructions), the “registers” in the Z-80 microprocessor, and a selected memory area that you are using for storage. You can also execute the assembly-language program at your own speed, from seconds per instruction to dozens of instructions per second, to allow you to follow the program step-by-step and see the changing conditions.

Along with the Assembly-Language Tutor Editor/Assembler/Interpreter we’ve also provided 25 Lesson Files. These are complete assembly-language programs, for the most part, that illustrate the text and make it easy for you to see examples of the techniques and instructions discussed in each lesson.

The text itself consists of 27 lessons, 25 of which are referenced to a Lesson File. Each lesson contains practical examples of assembly-language programming, discussion of Z-80 instructions, and student exercises.

The Assembly-Language Tutor will teach you the most commonly used Z-80 instructions, show you how to write simple assembly-language programs, teach you how to “interface” these programs with BASIC to enhance BASIC processing with the high speed of assembly language, and will give you guidance in planning larger assembly-language projects.

Special thanks are due Craig Verbeck for his help in debugging. Thanks also to my wife Janet for her debugging help and, above all, her patience.

How to Use This Course

Although this course is meant as a prerequisite to use of the Series I Editor/Assembler, it is not necessary to have any additional software. You should have a Model I with at least 16K of RAM and Level II BASIC or a Model III with at least 16K of RAM and Model III BASIC; at least one disk drive is required.

The lessons are about equal in length. Plan on completing one lesson at each sitting, if you can. In most of the lessons you’ll be asked to perform exercises, either in following existing programs from the Lesson

Files, or in writing your own short program segments. Do these exercises if you can, as this is the fastest way to learn to program in assembly language.

A dashed line following text signifies that you should perform the indicated action before continuing. In some cases this will be thinking through a simple question. In other cases it will be writing a short program segment.

Assembly-language programming is fun, challenging, and can greatly supplement the flexibility of BASIC. Good luck! See you at the end of the course!

Table of Contents

Lesson 1. How to Use the ALT

What is ALT? — How to Load ALT — Loading Lesson Files — What the Display Represents — Commands

Lesson 2. Z-80 Registers

What is a Register? — Register Functions — Working in Binary and Hexadecimal — Using the Modify Register, Trace Memory, and Modify Memory Commands

Lesson 3. Assembly Language

The Instruction Set — Instruction Mnemonics — Comment Lines — Labels — The Assembly Process — Executing the Program — Other Commands — Using the Editor, Assembler, and Interpreter

Lesson 4. Loading Registers

Loading 8-Bit Registers — Immediate Loads — Loading 16-Bit Registers — Transferring Data Between CPU Registers

Lesson 5. Loading and Storing Between CPU Registers and Memory

Eight-Bit Direct Loads and Stores to Memory — Eight-Bit Indirect Loads and Stores — Sixteen-Bit Loads and Stores — IX and IY Used as Indirect Registers

Lesson 6. Adding and Subtracting 8- and 16-Bit Numbers

Eight-Bit Adds — Sixteen-Bit Adds — Eight-Bit Subtracts — Two's Complement Numbers

Lesson 7. Adds, Subtracts, and Flags

More on Adds and Subtracts — The Z Flag — The S(ign) Flag — The Compare — The P(arity)/O(overflow) Flag — The C(arry) Flag

Lesson 8. Loops, Unconditional Jumps, and Conditional Jumps

Symbolic Addressing — Unconditional Jumps — Conditional Jumps — Conditions for JP — A Comparison Test Using Modify Memory

Lesson 9. Relative Jumps, Conditional and Unconditional

Using a JR in Place of a JP — Relative Addressing — Types of JRs — Limitations of JRs — A Special Relative Jump

Lesson 10. Other Arithmetic and Logical Operations

Increments and Decrements — The NEG and CPL Instruction — Logical Operations — A Sample Program Using ANDs and ORs (or ORs or ANDs)

Lesson 11. Subtracts with Carry and Multiple Precision

Multiple-Precision Numbers — Eight-Bit Add with Carry — Sixteen-Bit Add with Carry — Eight-Bit Subtract with "Borrow" — Sixteen-Bit Subtract with Borrow — Other Multiple-Precision Operations

Lesson 12. How to Move a Block Away

Block Move Number 1 — Changing the Number of Bytes Moved — Changing the Source and Destination Pointers — How the Program Works — A Better Block Move — Good, Better, Best... — How to Get in Trouble with an LDIR — The LDDR

Lesson 13. Table Techniques Part I

A List of Data — A Numeric Table Lookup — The DEFB Pseudo-Op — Entries of More Than One Byte — The DEFW Pseudo-Op — The DEFM Pseudo-Op — Accessing Multiple-Byte Table Entries

Lesson 14. Table Techniques Part II

Indexed Addressing — Table Operations Using Indexing

Lesson 15. Table Techniques Part III

Types of Orders — Sorting — Using the Carry Flag for Comparisons — A Bubble Sort of a Two-Byte Entry Table

Lesson 16. Block Compares

The CPIR Instruction — The CPDR Instruction — Using CPIR and CPDR to Scan a Table — CPI and CPD Operation

Lesson 17. Subroutine Use

Subroutine Basics — The Stack — Nested Stack CALLs — A CALL for Every RET — Types of CALLs and RETurns

Lesson 18. Other Stack Operations

Stack Uses — PUSHes and POPs — Multiple Subroutines

Lesson 19. Shifting Data

Rotates — RLD and RRD — Other

Lesson 20. More Shifting and Multiplication

Logical Shifts — Multiplying and Dividing by Shifting — Software Multiplies — A 16 by 8 Multiply — Arithmetic Shifts

Lesson 21. Bit Operations and Divides

The BIT Instruction — The SET Instruction — RES Instruction — Flags for the BIT, SET, and RES Instructions — A Divide Using SET, RES — Dividing by Larger Numbers — Doing “Signed” Multiplies and Divides

Lesson 22. Bits and Pieces

The Decimal Instructions — Using the Second Register Set — The NOP Instruction — HALT Instruction — Interrupt-Related Instructions — IN and OUT Instructions — RST Instructions — Exchange (SP) Instructions — What Do You Do With All the Instructions?

Lesson 23. Interfacing with BASIC — Linkages

Memory Maps — The ORG (Origin) Command — Relocatability — Transferring Control to an Assembly-Language Program — Loading the Object — Defining Where the Object Is to BASIC — Transferring Control to the Machine-Language Code — Executing CLRSCN

Lesson 24. Interfacing with BASIC — Passing Parameters

Passing a Parameter — Passing a Parameter Back — A Sample Parameter-Passing Subroutine — Running the Subroutine in BASIC

Lesson 25. VARPTR and Passing Multiple Arguments

VARPTR — Using VARPTR With Arrays — Passing Multiple Arguments

Lesson 26. Using ROM Subroutines

Cautions on Using ROM Subroutines — A Simple Text Editor — Using ROM Subroutines for Your Own Code

Lesson 27. Where Do You Go From Here?

Program Design — Program Flowcharting — Program Coding — Program Debugging — Program Documentation

Appendix I. ALT Commands

Appendix II. ALT Assembler Pseudo-Ops

Appendix III. Binary/Decimal/Hexadecimal Conversions

Appendix IV. Conversion Techniques

Appendix V. Z-80 Instruction Set

Appendix VI. Two's Complement Numbers

Appendix VII. Printable ASCII Codes



Lesson 1

How to Use the ALT

What is ALT?

ALT stands for "Assembly-Language Tutor." We abbreviated this to ALT. ALT is an assembly-language program that will teach you assembly language. It contains:

- An editor, similar but not identical to the BASIC program editor. This portion of ALT will help you construct text lines that represent the "language" portion of assembly-language programs. A typical line might be:

```
100 START      LD          A,23          ;load A reg with 23
```

This particular line is eventually decoded as an instruction to the computer that says "At a location called START we'll put an instruction to load a value of 23 into the A register." No need to worry about what that instruction does now; just observe that you can talk to the computer via assembly language by plain text, such as line 100 above.

- An assembler. This portion of ALT takes a sequence of assembly-language lines like the one above and converts them into numbers that the microprocessor can recognize. This conversion process is called "assembly." The line above, for example, would be converted to the decimal numbers of

```
3E 17
```

which, when input to the microprocessor in your Model I or III would cause a value of 23 to be put into a type of memory location called A.

- A "debug" package. This portion of ALT will let you "execute," or run, your assembly-language programs. It will let you look at certain memory areas in which results are stored, change the areas at will, and generally let you control things.
- An interpreter. This portion of ALT is the "controller." Assembly language is so fast that it's hard to see what is happening. The interpreter interprets each assembly-language instruction and slows it down to anywhere from dozens of instructions per second to one instruction every few seconds.

At the same time, the interpreter displays the current instruction on the screen so that you can follow exactly where you are in the program and what is happening inside the microprocessor and memory.

Another important facet of the interpreter is that it doesn't allow you to lose control by putting in incorrect assembly-language instructions. You can't "blow-up" your programs, as the interpreter will tell you that you've made an error.

All the parts of the ALT are in one package. Once you load it, you don't have to switch between different parts, but have complete control of the assembly-language program.

There's another part of ALT also. You'll find a disk with your system. On the disk is a series of 26 lessons. These lessons contain complete assembly-language programs that correspond to each lesson in the text.

You can proceed at your own pace with the lessons; we would recommend, however, that you try to complete each lesson at one sitting.

How to Load ALT

The first step of every lesson is to load the ALT program. The ALT program is on either cassette or disk, depending upon the type of system that you have.

You must have at least 16K of "RAM" (random-access memory) and Level II BASIC (Model I) or Level III BASIC (Model III) to be able to use ALT.

For disk-based systems:

Load TRSDOS as you would normally so that you get the

1 How to Use the ALT

TRSDOS READY
prompt message.

Type

ALT followed by ENTER

ALT should now start executing, and you should see a title message and “menu” of items. Now sharply hit ENTER, and you’ll see the display of Figure LESS1-1.

```
PC  AH AB      S Z H P N C B C D E H L IX  IY  SP
FFFF FF 11111111 1 1 1 1 1 1 FF FF FF FF FF FF FFFF FFFF FFFF
.....
STATUS LOCN CONTENTS  LINE  LABEL  OPCODE  OPERAND COMMENT
.....
LOCN 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
.....
MODE=EDIT SPEED=9000 FREE MEMORY=21360 BP @ FFFF,FFFF,FFFF,FFFF
COMMAND=
```

Figure LESS1-1. ALT Basic Display

Load ALT now so that you get the display shown above. After the load, go on to the next discussion.

Loading Lesson Files

At the beginning of each lesson, you will be asked to load the Lesson File for the lesson. These Lesson Files all have the same type of name

LESSXX

where “XX” is the number of the lesson. The file for Lesson 20, for example, is called

LESS20

To load a Lesson File from disk:

- Enter

L followed by a space, followed by the Lesson name, such as LESS20, followed by ENTER.

- You should see the disk light go on, and the file should load.

At the end of the load, you should see text in the middle of the screen, as shown in Figure LESS1-3.

```
PC  AH AB      S Z H P N C B C D E H L IX  IY  SP
FFFF FF 11111111 1 1 1 1 1 1 FF FF FF FF FF FF FFFF FFFF FFFF
.....
STATUS LOCN CONTENTS  LINE  LABEL  OPCODE  OPERAND COMMENT
.....
00099 ;.....LESSON 4.....
00100 : ADC FOR MULTIPLE-PRECISION 4-BYT
00110 MPADDS LD      HL,0B003H
00120      LD      IX,0B007H
00130      LD      B,4
.....
LOCN 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
.....
MODE=EDIT SPEED=9000 FREE MEMORY=19025 BP @ FFFF,FFFF,FFFF,FFFF
COMMAND=L LESS1
```

Figure LESS1-3. Lesson File Display

Load Lesson File Number 1 by following the steps above. The name of this Lesson File is LESS1.

You should now have the screen display shown in Figure LESS1-3.

What the Display Represents

There are 5 areas of the display, as shown in Figure LESS1-4.

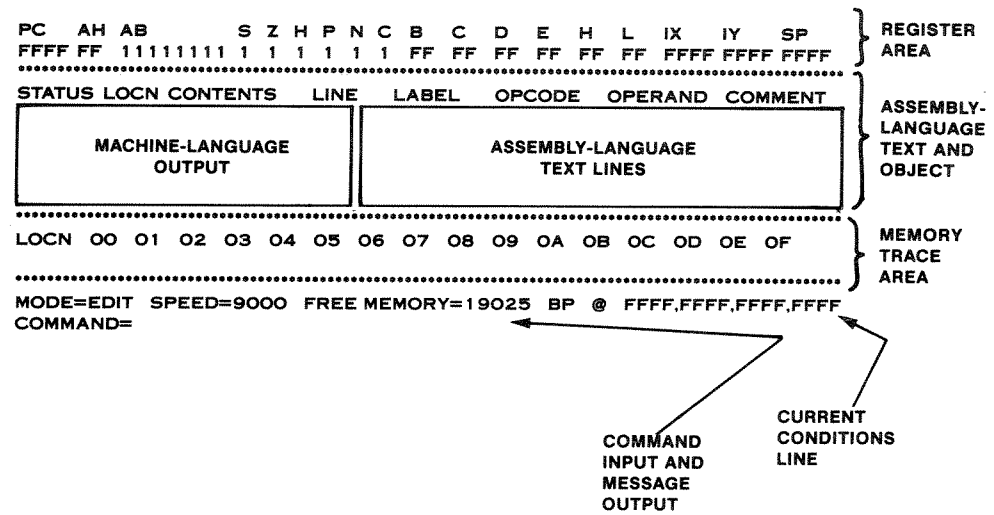


Figure LESS1-4. Display Areas

The first “band” on the screen shows the Z-80 microprocessor “registers.” The registers are high-speed memory locations that are located not in RAM memory, but inside the microprocessor itself. The microprocessor is the small “chip,” or integrated circuit, that is the heart of all the “logic” of your system.

We’ll be working continually with these registers in assembly language. We’ll describe them in detail in the next lesson.

The register names are shown on the top line. On the next line, the register “contents” are shown, directly below the corresponding register names. You’ll see the contents line change as the program executes or as you modify the contents of the registers under ALT.

The next “band,” after the asterisks, is a text area of 6 lines. The first line is fixed and is a heading for the 8 “fields” of STATUS, LOCN, CONTENTS, LINE, LABEL, OPCODE, OPERAND, and COMMENT.

The right portion of the text area shows the assembly-language text lines. This is the text that is in the Lesson File or that you have entered for an assembly-language program.

Just as in BASIC, each line of text has a “line number” associated with it.

You should see the text for Lesson 1 displayed on the screen at this point.

The left portion of the text area normally displays the “output” of the assembler portion of ALT. The lines in this area correspond to the lines of text. They are the “machine-language,” or microprocessor code, equivalent of the text lines. At this point, you should not see anything in the left portion of the text area.

The next “band” after the asterisks is a memory “trace” area. This area displays the contents of memory locations that you select.

To see how it works, type

ZT 8000 followed by **ENTER**

You should see the display shown in Figure LESS1-5, which represents the 8000 hexadecimal memory area of RAM (the values may be different).

1 How to Use the ALT

```
PC  AH AB      S Z H P N C B C D E H L IX  IY  SP
FFFF FF 11111111 1 1 1 1 1 1 FF FF FF FF FF FFFF FFFF FFFF
.....
STATUS LOCN CONTENTS  LINE  LABEL  OPCODE  OPERAND  COMMENT
```

STARTING LOCATIONS

```
.....
LOCN 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
B000 53 02 10 05 23 FF FF FF 0B AB C2 83 95 FE 28 DD
B010 09 11 08 7A EF E2 F0 18 42 95 87 88 E2 83 15 23
.....
MODE=EDIT SPEED=9000 FREE MEMORY=21360 BP @ FFFF,FFFF,FFFF,FFFF
COMMAND=
```

NOTE: DATA SHOWN
IS TYPICAL. DATA ON
YOUR DISPLAY WILL
BE DIFFERENT

THIS AREA DISPLAYS
THE LOCATIONS B000H
THROUGH B01FH.

Figure LESS1-5. Memory Trace Area

The next band, after the asterisks, contains 2 lines.

The first line contains the current “conditions” under which ALT is operating.

The MODE is either EDIT for editing mode, ASSM for assembly mode, or EXEC for execute (interpreter) mode.

The SPEED is a number from 0 to 9999. This represents the speed at which programs will execute and is controlled by you. A value of 0 is a low speed, while a value of 9999 is high speed. Low speed is about one instruction every 3 seconds, while high speed is dozens of instructions per second.

FREE MEMORY tells you how much memory you have left. This will change as you enter text and do other operations, such as assemblies.

The BP @ message indicates the “breakpoints” that you have set. A Breakpoint is a special debugging device that lets you stop the program at selected instructions. You may have 4 breakpoints at any time.

The last line on the screen starts with COMMAND=. This is the line in which you’ll be entering all ALT commands. As you type the command, you’ll see it appear to the right of the COMMAND= message.

The right-hand portion of the command line is used for error messages, such as DISK ERROR or INVALID ARGUMENTS. The error message will disappear with the next command that you enter.

Commands

The complete command summary for ALT is given in Appendix I. We’ll go through some of them here.

Every command is terminated by an ENTER, and we won’t repeat the ENTER when we’re describing the following commands.

Displaying Text by the P Command

Edit Mode commands are similar to BASIC Edit Mode commands, but are not identical.

You should have text displayed from loading the Lesson File.

Printing Lines

To display the end of the text, type

P.

You should see the last line of Lesson 1, line 440. The * character is a special character that stands for the last line of the text.

To get back to the first line of text, type

P#

You should see the beginning of the text. The # character is a special character that stands for the beginning line of the text.

To display the text from any line, type

PLLL

where LLL is the line number. To display from line 200 on, for example, type

P200

You should see the text from 200 on displayed.

A special form of the P command lets us “scroll through” the text. Entering

P

alone, displays the “current text.” Entering one P after another displays the next 5 lines of text at a time. Try it now.

To get back to the start, end, or any given line, just use the P#, P*, or PLLL form of the command.

Deleting Lines

You can delete lines from the text file by using the D command. To delete line 100, for example, you’d type

D100

Get back to the beginning of the text by typing P#. Do a delete of line 100 by D100 now.

You should see line 100 disappear and the remaining lines “scroll up.”

Another form of the command is

DNNN:MMM

This form deletes lines NNN through MMM. Try to delete lines 120 through 140 now.

Did you use

D120:140?

If so, you saw the lines deleted and a scroll to the next lines of text.

Inserting Lines

The third Edit Mode command lets you insert text. There are two conditions for this insert command.

If you do not have text in the “buffer” you can use the I command to create new text. You’d want to do this if you were creating a new assembly-language program, for example.

To see how this works, delete all the text in the buffer by

D#:*

1 How to Use the ALT

You should see the text disappear from the screen entirely.

Now type

1100,10

You'll see a "100" under the LINE heading appear. You can now type anything you'd like (at least in this lesson). Go ahead, type until you get to the end of the line.

If you typed text until the end of the line, you saw a new line number appear — line 110. ALT automatically terminates a line if more than 34 characters are entered.

The 100 in the I command says "start at line 100." The 10 says "increment by 10." You could keep adding text up to the limits of the text buffer. You can use any start number and increment you'd like (111 and 13, for example), but the "normal" ones often used are 100 and 10.

To end any line just type ENTER.

Try using the Insert mode for a while. When you want to get back to the next Command, press BREAK. BREAK stops the insert mode and returns you to the ALT Command Mode for the next command.

BREAK will return to the ALT Command Mode under most conditions.

Type Q (Quit) while in Command Mode to return to TRSDOS.

To Sum It All Up

At the end of each Lesson, we have a little section entitled "To Sum It All Up." It will review what you've learned in the lesson.

In this first lesson we've covered:

- ALT is a program that consists of an editor, assembler, interpreter, and debugging package
- ALT allows you to execute assembly-language programs under your control and has preventative measures in it to keep the program under control
- The display for ALT has 5 areas that display the contents of registers in the Z-80 microprocessor, the assembly-language text and machine language for the program, a selected memory "trace" area, the current ALT mode and conditions, and the current command
- The P, D, and I Edit Mode commands let you display, delete, and insert lines of text for the program
- The BREAK key will usually return you to the Command Mode

For Further Study

Appendix I ALT Commands

Lesson 2

Z-80 Registers

There is no Lesson File for this lesson.

In this lesson we're going to look at the Z-80 microprocessor "registers." We'll also find out how to change them through ALT.

What Is a Register?

By this time you probably know what RAM (random-access-memory) and ROM (read-only-memory) are, but we'll briefly describe them anyway. RAM is a "read-write" memory organized in **bytes**, which are 8 **bits or binary digits** long, as shown in Figure LESS2-1. ROM is a "read-only" memory also organized in bytes, as shown in the figure.

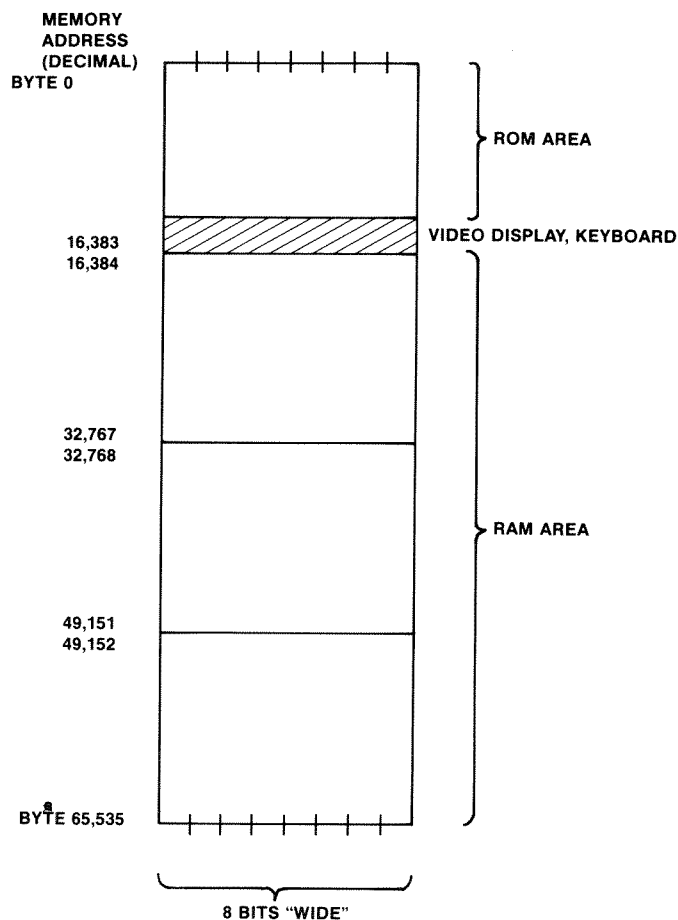


Figure LESS2-1, ROM and RAM

A bit is either a 0 or a 1, on or off, lighted or unlighted. A typical byte of 8 bits, therefore, might be something like 10110111 — 8 different bits of any combination.

The Z-80 microprocessor in the Model I or III contains a group of "registers," which are nothing more than memory locations, very similar to RAM.

Unlike RAM and ROM, though, which are addressed by a location value of 0 through 65,535, microprocessor registers are called by letter designations, as shown in Figure LESS2-2.

2 Z-8 Registers

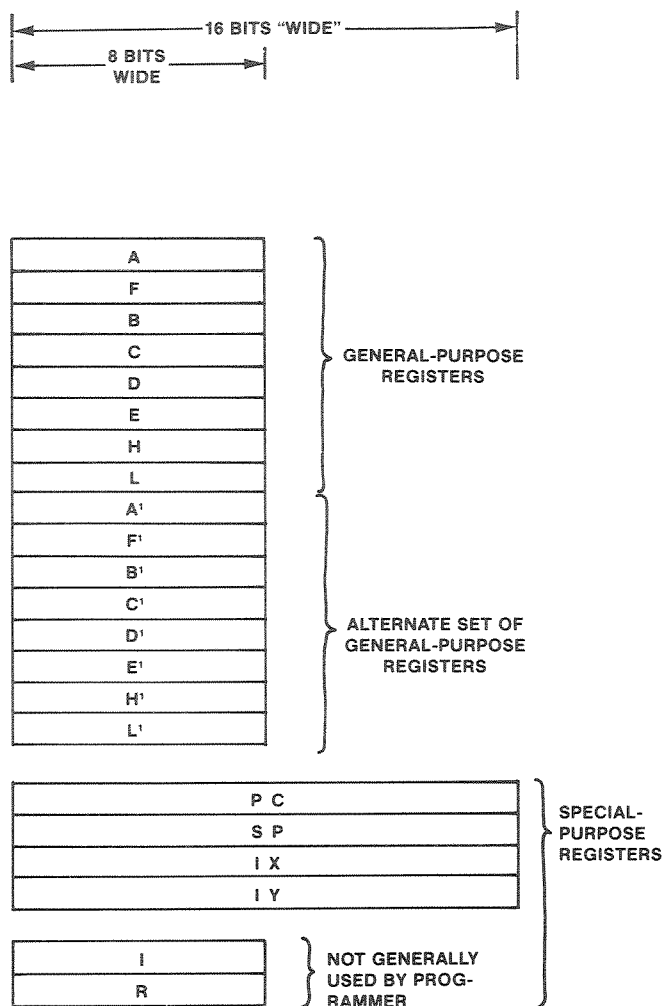


Figure LESS2-2, Z-80 Registers

The letter designations are not necessarily related to their functions, although some are.

In general, Z-80 registers, or "cpu registers," are used to hold the results of temporary operations. Because they can be read from or written to at faster speeds than RAM memory, using cpu registers rather than RAM memory locations speeds up the microprocessor instructions, which speeds up any program.

The entire "instruction set" of the Z-80 and other microprocessors is geared to using the cpu registers. Although there are many instructions which handle reading and writing to memory, just about all data passes through one or more of the cpu registers.

Register Functions

Look at Figure LESS1-1. On the top line, you'll see the listing of all of the cpu registers.

All register contents on the second line are shown in **hexadecimal**, a shorthand way of representing binary. There are two hexadecimal digits for every 8 bits, so you'll see 2 hex digits under many of the registers. The A register is also shown in binary.

Some of the registers hold 2 bytes instead of 1 byte, however, twice as much as other registers or RAM locations. In this case you'll see four hex digits.

Hex digits are 0 through 9 and A through F. We'll explain them in a moment.

The first register is the PC, or Program Counter. This is the 2-byte register in the cpu that “points to” the next microprocessor instruction in memory.

Next are “AH” (A register in hexadecimal) and “AB” (A register in binary). The AH display has 2 hex digits under it, as the A register is 8 bits. The AB is the same value in binary, a total of 8 binary digits or bits. Note that the bits are either 0 or 1.

Next are 6 special bits called **Flags**. The flags are S, Z, H, P, N, and C. These flags all have special meanings that we’ll discuss in later lessons. For now, just note that there are 6 of them, and each holds either a 0 or 1. The Flags are collected together in a special register called the F register.

Next are B, C, D, E, H, and L. These are six additional registers in the cpu — each one 8 bits or two hex digits.

The last three registers in the cpu are the IX, IY, and SP registers. Each of these are 16 bits, or 2 bytes, and are represented by 4 hex digits.

In addition to these registers, there is a duplicate set of the A, F, B, C, D, E, H, and L registers called the “prime” registers. These were shown in Figure LESS2-2. Only one set of these registers, prime or non-prime, can be active at any time. We will see later how to switch between them in programs.

Working in Binary and Hexadecimal

If you want to do assembly-language programming, you’ll have to become a little familiar with binary and hexadecimal notation. You won’t have to become a math whiz at it, but you will have to be able to convert between decimal numbers and binary and hexadecimal.

Let’s consider 8- and 16-bit binary numbers, since these are the ones we’ll be working with the most. Each 8-bit number represents data as shown in Figure LESS2-3.

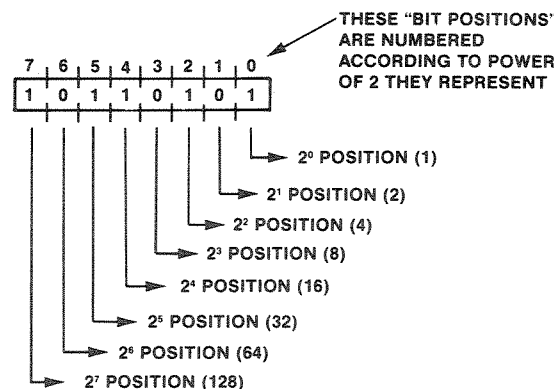


Figure LESS2-3. Eight-Bit Binary Numbers

Each digit position of an 8-bit binary number represents a power of two. The “bit position” on the right is 2 to the zero power, or 1. The next is 2 to the first power, or 2. The next position is 2 to the second power, or 4. The next positions are 8, 16, 32, 64, and 128.

To find the equivalent decimal number represented by an 8-bit binary number, just add together the powers of two represented:

$$\begin{array}{r}
 10101100 = ? \\
 128 \\
 +32 \\
 +8 \\
 +4 \\
 \hline
 10101100 = 172 \text{ decimal}
 \end{array}$$

2 Z-8 Registers

To convert from decimal to binary, find the powers of two that make up the decimal number. To convert 135 to binary, for example:

135 = ?	
Does 128 "go" into 135? — yes,	135-128=7
Does 64 "go" into 7? — no	
Does 32 "go" into 7? — no	
Does 16 "go" into 7? — no	
Does 8 "go" into 7? — no	
Does 4 "go" into 7? — yes,	7-4=3
Does 2 "go" into 3? — yes	3-2=1
Does 1 "go" into 1? — yes	1-1=0
135 = 128+4+2+1 = 10000111 in binary	

If this all seems confusing, don't despair. We have an appendix to convert between decimal, binary, and hexadecimal in the back, Appendix III. It will convert all values from 0 through 255.

Before you look at the Appendix, though, try some values yourself:

- What is 01110010 binary in decimal?
 - What is 10101010 binary in decimal?
 - What is 255 decimal in binary?
 - What is 33 decimal in binary?
-

Did you get the answers without looking in the Appendix? The answers are

- 01110010 = 114 decimal
- 10101010 = 170 decimal
- 11111111 = 255 decimal
- 00100001 = 33 decimal

When you're using 16-bit binary numbers, the process is the same, but the additional bits represent larger powers of 2, as shown in Figure LESS2-4. The "high-order" bits represent 32768, 16384, 8192, 4096, 2048, 1024, 512, and 256. The "low-order" 8 bits represent values as in 8-bit numbers. We won't burden you with a lot of exercises, but we'll just show you one conversion:

1010111100001111 = ?	
	32768
	8192
	2048
	1024
	512
	256
	8
	4
	2
	1
	<hr/>
	44815

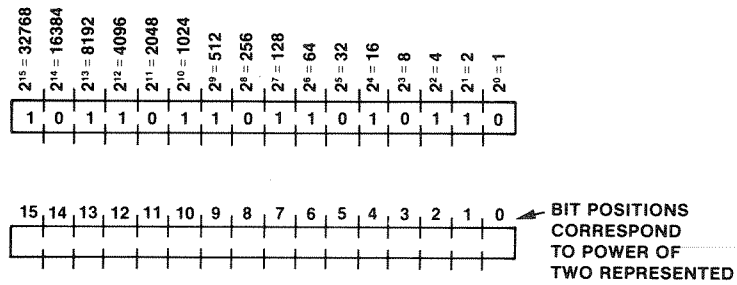


Figure LESS2-4. Sixteen-Bit Binary Numbers

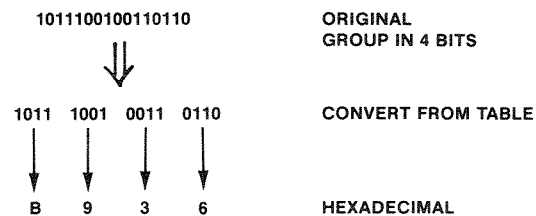
You can see that it gets rather tedious to represent long strings of binary numbers. Programmers use a kind of shorthand to make things more compact. The shorthand is called “hexadecimal.”

The hexadecimal numbers and their decimal equivalents for 0 through 15 are shown here

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

To convert any binary number to hexadecimal, simply divide into groups of 4 bits and use the chart above to find the hex equivalent of each group, as shown in Figure LESS2-5.

CONVERTING FROM BINARY TO HEXADECIMAL



CONVERTING FROM HEXADECIMAL TO BINARY

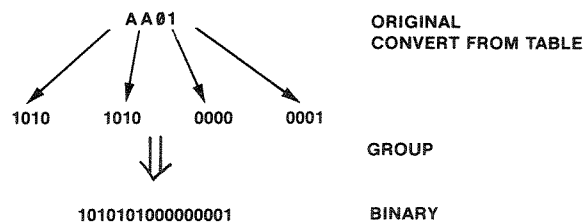


Figure LESS2-5. Binary/Hexadecimal Conversions

2 Z-8 Registers

To convert back again, translate each hex digit to a 4-bit binary value, as illustrated in the figure.

The easiest way to convert between the three types of numbers is to use Appendix III. After a while, you'll be able to work well with binary and hex numbers because you will see them frequently.

Hexadecimal numbers usually have an "H" suffix to indicate that they are in hex. The number 0100H, for example, is 100 hex, or 256 decimal (100H=000100000000 binary).

Hexadecimal numbers often have a "leading" 0 if the number starts with A through F. The reason for this is that many programs cannot decide whether a number like FACE is a hexadecimal number or a text name. A 0FACE leaves no doubt.

Using the Modify Register, Trace Memory, and Modify Memory Commands

To give you some experience in binary and hex, we'll show you the ZR, ZT, and ZM commands. These commands are special ALT commands that are entered in Command Mode.

ZR — Modify Register

This command lets you modify a cpu register. The form is

ZR X=VVVV

where X is a register name of A, F, B, C, D, E, H, L, AF, BC, DE, HL, IX, IY, SP, or PC. The VVVV is a hexadecimal value of 1 through 4 hex digits.

Enter

ZR A=11

You should see the A register change to 00010001 under AB and to 11 under AH. Try some other values to see the changes. Try entering values from 0 through A, and watch how the binary representation of A (AB) changes from 00000000 through 00001010.

Try changing the other registers. Note that some registers are grouped together, like AF, BC, DE, and HL. These registers are called **register pairs** and when taken together this way represent a 16-bit register instead of two 8-bit registers. Note that when you change the HL register pair by

ZR HL= 1234

both the H and L registers change, but that you can also change each individually by

ZR H=12 and ZR L=34

ALT will give you an "INVALID ARGUMENTS" message if you try to enter too many hex digits or something other than a hex digit. In that case, just try again.

ZT — Trace Memory

The ZT command lets you display any 32-byte area of memory. The format of ZT is

ZT MMMM

where MMMM is a 1- to 4-digit hex value.

Enter

ZT B000

You should see B000H and B010H under the LOCN column in the trace area. There will be 16 hex values on the first line and 16 hex values on the second line in this area.

The "00" through "0F" on top represents the last portion of the address. To find location B01EH, for

example, you'd look in the B010H row and then under the 0EH column to find the value for that memory location.

The LOCN columns always start with a number that is a multiple of 16. These numbers are numbers like B000H, B010H, B020H, and so forth. ALT always "rounds off" any location value input to this type of number. For example, try

ZT 7111

You'll see the display change to 7110 and 7120 under the LOCN column.

ZM — Modify Memory

You can change memory locations just like you were able to change register values by using the ZM command.

The format of the ZM command is

ZM HHHH=

where HHHH is a memory location in hexadecimal.

Enter

ZT B000

and then enter

ZM B000=

ALT will reply with something like

ZM B000=00

The 00, or whatever value is displayed, represents the current contents of that memory location. To change the value, simply enter a new value. To leave the contents of the memory location unchanged, enter a space alone without any value.

Suppose you wanted to change locations B000H through B005H to 12H, 34H, 56H, 78H, 9AH, and BCH, for example. You'd have something like

ZM B000=XX 12
B001=XX 34
B002=XX 56
B003=XX 78
B004=XX 9A
B005=XX BC

Try it, and you'll see the memory data changing on the trace display.

To get out of the memory change mode, just hit ENTER before any value.

Experiment with the ZM command in the B000H area of memory.

ALT will allow you to change any area of memory except:

- The ALT program
- The text area
- Any area used to hold the machine-language program after assembly

If you get a "NOT DATA LOCATION" error, you'll know that you're out of a data area. By the way, this is done, believe it or not, to keep the ALT program and any program you might enter, under control, and not for reasons of secrecy!

2 Z-8 Registers

To Sum It All Up

To review what we've learned in this lesson:

- A register is a special fast-access memory location in the microprocessor used to hold temporary results
- Registers are either one byte (8 bits) or two bytes (16 bits) long
- The registers are named A, F (Flags), B, C, D, E, H, L, IX, IY, SP, and PC
- There is a duplicate “primed” set of the A, F, B, C, D, E, H, and L registers
- Some registers may be grouped together as “register pairs” — AF, BC, DE, and HL
- Binary notation represents data with binary digits of 0 and 1
- Bit positions represent powers of two starting with 2 to the zero power (1) on the right and increasing to the left
- Numbers can be converted from binary to decimal by adding together the “weights” of 128, 64, 32, etc.
- Numbers may be converted from decimal to binary by seeing which powers of 2 “go” into the decimal number
- Hexadecimal representation is a shorthand notation for binary numbers
- Hexadecimal digits are 0 through 9 and A through F
- To change from binary to hex, convert 4 bit groups into hex digits. To reconvert, reverse the procedure
- The ZR command lets you change register values
- The ZT command lets you display memory areas
- The ZM command lets you modify memory locations, as long as those locations are not being used by ALT or user programs

For Further Study

Appendix I ALT Commands

Appendix III Binary/Decimal/Hexadecimal Conversions

Appendix IV Conversion Techniques

Lesson 3

Assembly Language

Load LESS3 from disk.

In this lesson we'll look at what assembly language is, and how an assembler converts an assembly-language program to machine-language instructions.

The Instruction Set

The Z-80 microprocessor used in the TRS-80 Model I and III has a built-in "instruction set." This instruction set consists of dozens of different types of instructions.

Computers started out as fast adding machines, and for that reason, a lot of the instructions are oriented toward arithmetic operations. Instructions that add two numbers or subtract two numbers are common.

There are other types of instructions that are related to program flow — such instructions as "Jump to a Location" and "Jump if a Zero Result."

Part of the problem in learning assembly language is in memorizing the different instructions, their effect, and their formats.

All of the instructions of the Z-80 taken together are called the "instruction set" of the Z-80. The instruction set of the Z-80 is really the instruction set of the Model I and III as well, as no new instructions have been added by Radio Shack.

Instruction Mnemonics

It's much easier to say `ADD E` than to say "Add the contents of the E register in the cpu together with the contents of the A register and put the result in the A register." Lengthy instruction descriptions are simply denoted by an **instruction mnemonic**. There are two parts to the mnemonic, the operation and the operands.

If you have loaded Lesson 3, you should have a display of the first part of a typical program on the screen in the text area. Look at some of the text under the `OPCODE` column.

What you're seeing there is the operation mnemonic for the instruction. This is sometimes called the **op code**, for "operation code." The opcode `LD` in the second line stands for Load. The opcode several lines down (use the P command to scroll), the `CP`, stands for Compare.

The second part of the instruction mnemonic is the operands portion. This is sometimes called the argument portion. The operand and op code, taken together, define what the instruction does.

The operands for the `LD` in the second line are `E,0`, which, when taken together with the `LD`, stands for "Load the E register with a value of 0."

The `CP` line really means "Compare the contents of the A register with a memory byte at an address pointed to by the contents of the IX register plus 1." (Quite a mouthful, eh? Don't worry about what the instructions do at this point. We'll be getting into that soon enough!)

Comment Lines

Any line that starts with a semicolon is simply a comment line and is not treated as an instruction. Any text in the last part of the line that has a semicolon before it is considered a comment, also.

You can either load the Lesson File for the programs in the following lessons, or you can key in the programs shown in the lessons for practice. If you do the latter, you don't have to put the comment lines or comment "fields" into the program. They will not affect the program.

3 Assembly Language

Labels

The BUBSRT, BUB010, and BUB020 names are called “labels.” A label is used in assembly language in lieu of a line number as in BASIC. They are equated to the memory location at which the instruction is stored. The LD E,0 instruction, for example, might be stored in RAM memory location 0C000H, but the label BUBSRT would be the label of the instruction during assembly.

The Assembly Process

The entire collection of text that defines Z-80 instructions and comments constitutes an “assembly-language” program. What do we do with it? How do we feed it into the computer?

The Z-80 microprocessor cannot accept text and decode it. Maybe the next generation of microprocessors will. We have to take the text representing the assembly-language program and convert it into a form that the Z-80 can understand — binary ones and zeroes.

This process is called “assembly.” The ALT has a built-in assembler that will translate the text of the program in “assembly language” into “machine language,” the binary ones and zeroes.

The command for this is

A

for Assemble. Enter an A now for the command.

You should have seen a rapid display of the text in the assembly-language portion “scrolling up” on the screen. At the end, you’ll see something that looks like Figure LESS3-1.

```

THESE LOCATIONS
MAY BE DIFFERENT

PC  AH  AB           S  Z  H  P  N  C  B  C  D  E  H  L  IX  IY  SP
FFFF FF 11111111 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 
```

Figure LESS3-1. Sample Assembly

The portion on the left of the text area represents the machine-language output of the assembler. This is sometimes called “object language” and the assembly-language text is sometimes called “source language.”

The hexadecimal values under the CONTENTS column are the actual hexadecimal codes, which, when converted to binary values, represent the codes for each instruction in the assembly-language program.

What have we really done at this point? We’ve really only used a program to translate a more English-like form of a program into binary ones and zeroes, the program being the assembler in ALT.

The machine-language form is on the screen. The actual machine-language values are also in memory at this point, and the memory locations where they are stored are shown in the LOCN column.

The LOCN column in the second part of the screen shows a hexadecimal RAM location for the machine-language program. Note that the locations do not increment by one for each instruction. That’s

because the machine-language codes vary in size from 1 to 4 bytes in length. The LD E,0 instruction, for example, is 2 bytes long in machine-language form. The CP (IX+1) is 3 bytes long.

ALT is different from some assemblers that do not put the machine code in memory after assembly. In these assemblers the “object code” goes onto a cassette or disk file.

Note that there is generally a “one-for-one” correspondence between an assembly-language form of the instruction and the machine-language form of the instruction. One machine-language instruction is generated for each assembly-language instruction. Comment lines are ignored and do not generate any object code, along with certain other types of assembly-language text.

The A command can be used to assemble any assembly-language program you have in ALT. You may create your own program, using the Edit Mode Insert and Delete commands, or you can assemble existing assembly-language programs from the Lesson File.

If you have a line printer on your system, you can use the

A LP

form of the Assemble command to get an assembly-language **listing** of the program in ALT. Try this form of A right now, and you should get a printout corresponding to the screen display.

Another form of A is

A WE

This will halt the assembly if any assembly errors are found, with the type of error displayed under the STATUS column for the appropriate instruction.

Executing the Program

Once you have assembled a program you can execute it by using ALT. ALT does not allow execution of a program directly. It looks at each machine-language instruction in memory, decides what it does, and then simulates the execution exactly. This means that the execution will go at a slow rate, but that ALT will be easily able to detect when you have made errors in the program that would cause the program to jump out of the program area or to cause data to destroy other instructions in the program.

To execute any program under ALT, get an error-free assembly listing, and then do

ZX MMMM

where MMMM is a hexadecimal address corresponding to the location of the program (see the “LOCN” column in the fourth screen line). The address MMMM is optional. If you do not use it, the execute command will execute from the start of the current program in memory.

Try it for this program. To get the maximum effect, trace the B000H area by

ZT B000

assemble the program, and then execute it by finding the first address under the LOCN column and transferring control by the ZX command.

You should see the program execute. As it does, you’ll see the instructions (assembly language and machine language) “scroll by.” You’ll also see the registers changing values and the memory trace area changing data too. Stop by pressing the BREAK key.

A special form of the execute command, ZXS, lets you “single step” through each instruction. Try it now by ZXS MMMM. You’ll have to hit any key for every instruction to be executed.

3 Assembly Language

You can change the speed of execution by using the ZS command. The ZS command changes the execution speed. A 0 value is slowest, while a 9999 value is fastest. The format of ZS is

ZS VVVV

where VVVV is the 0 to 9999 value.

Try 0, 9999, and other values, and repeat the execution.

ZS is valuable because you can actually **see** the registers changing values as instructions are executed. You may wish to vary the speed depending upon the program. If you understand what a program does, then you'd run it at a high speed. If you did not understand certain parts of it, then you'd run it at a slower speed and observe the results.

The register display area shows the values in the registers while the program is executing. You can use the ZR command to change the values in the registers before execution or at certain other times.

Other Commands

Before we get into the discussion of actual instructions let's mention some other commands that we neglected before:

- The Q command simply takes you back to BASIC or TRSDOS, depending upon your system. Any program or data you have in the ALT buffer will be lost; you'll have to reload ALT to continue.
- The Hardcopy command lets you get a listing of the assembly-language portion of the text only (not the machine language) on your system printer. The format for Hardcopy is

HLLL:MMM

where LLL is the starting line number and MMM is the ending line number.

- The N command renumbers the assembly-language lines. The format of N is

NLLL,II

where LLL is the starting line number for the renumber and II is the increment. Doing a

N200,20

would renumber the current lines with the new lines starting at line number 200 and increasing by 20 for each line.

- Another form of the Insert (I) command lets you insert lines between any existing lines. The format of Insert for this purpose is

ILLL,II

where LLL is the line number for the insert point and II is the increment. If you had text line 100, 110, 120, 130, and so forth, and you wanted three new lines between lines 120 and 130, you might say

I121,1

The Insert mode would then be active and you'd see line 120 displayed in the text area, followed by the number 121. You could then enter the 3 new lines and press BREAK to exit. If you wished, you could then use the renumber command (N) to renumber.

If the Insert line number exists, the Insert mode will give an error message "LINE EXISTS." Inserts can be done until the line numbers increment up to an existing line number. If you run out of space, simply renumber by N and continue the Insert.

The W command lets you Write an assembly-language file to cassette or disk. The format of W is

W NAME

where NAME is the name of the file. Note that the file written is the assembly-language text only. It is not a file that can be run as a program. You can only use it to assemble under ALT. If you don't provide a name for the W command, the name "NONAME" is used.

A special form of the trace command, ZTT, displays a memory area in ASCII format. ASCII is a "text" display; any printable characters will be displayed as the characters. Any data that does not represent an ASCII character will be represented by a period.

The ZB and ZZ commands are used to set and clear (Zap) "breakpoints," execution "stop points." We'll cover breakpointing in a future lesson.

Using the Editor, Assembler, and Interpreter

You can switch back and forth between the Edit mode, assembler, and interpreter (execution controller) at will. However, editing the assembly-language text by Delete or Insert will destroy any machine-language program in memory, and you will have to reassemble by the A command.

In lessons to come we'll explain in detail what we're doing, so don't feel too badly if you feel lost at this point. We'll review the commands as we use them, and you should be an "old hand" after a few lessons.

To Sum It All Up

To recap what we've covered in this lesson:

- The instruction set of the Z-80 includes dozens of instructions, some relating to arithmetic operations and some relating to program control
- The instruction set is abbreviated by using mnemonics for the operation and the operands of each instruction
- Comment lines do not generate machine-language code
- Labels are used to represent an instruction by a symbolic form, rather than as an absolute location
- An assembler converts assembly-language code into the machine-language ones and zeroes that the Z-80 requires for executing instructions
- ALT executes machine-language instructions in memory by "interpretation" allowing complete control over the execution, but at slower speeds
- The Q command returns you to BASIC or TRSDOS
- The H command allows hardcopy printing of the text
- The N command renumbers the text
- Another form of Insert lets you insert assembly-language lines between existing lines
- The W command lets you write out an assembly-language file to cassette or disk

For Further Study

Appendix I ALT Commands



Lesson 4

Loading Registers

Load LESS4 from disk.

Up to this point we haven't done anything in the way of practical examples of assembly language. In this chapter we'll start doing some "useful work" with assembly language. Since we'll be continually using the cpu registers that we discussed in Lesson 2, we'd better get some practice in manipulating them. In this lesson we'll find out how to "load them" with data and how to move that data around within the cpu. (In the next chapter we'll see how data can be transferred between the cpu registers and memory.)

First of all, you might glance back at Lesson 2 to review the register "architecture," a fancy word for describing what registers are available for the assembly-language programmer and what their chief uses are.

The "general purpose" cpu registers are the A, B, C, D, E, H, and L registers. They're used to manipulate 8 bits of data at a time. The "main" register among these is the A register, which is an A(ccumulator) register. This is somewhat of an archaic term that means that this register is used to accumulate results of adds, subtracts, and other operations. We can do things with the A register that we can't do with the other general-purpose registers because there are special instructions that operate with the A register alone, such as "ADD A," which adds an 8-bit number to the contents of the A register.

Remember that the general-purpose registers can be grouped together to make up 16-bit registers. The B and C registers make up a 16-bit BC register, the D and E registers make up a 16-bit DE register, and the H and L registers make up a 16-bit HL register. The A register can also be grouped together with the F register, but this is for the convenience of "stack operations" (which we'll discuss in another lesson) and the A and F do not make up a true 16-bit register.

When the registers are grouped together as 16-bit registers, the HL register becomes like the A register. It is a special 16-bit "accumulator" that is the principle register used for arithmetic operations.

The remaining registers are not "general-purpose" registers, but are used for stack operations (SP), indexing operations (IX and IY), or program control (PC).

There are also the R and I registers, which are not used in simple programming, but which perform memory refresh operations (R) and interrupt-processing operations (I). We won't be discussing either of these operations in this book, as they are primarily "hardware" functions related to system operation.

Loading 8-Bit Registers

Let's first look at simple "loads" of the general-purpose 8-bit registers. Enter lines 100 through 300 of the following code (reload ALT to get a "blank screen" and then use the I command to start the entry), and assemble, or use Lesson 4 directly from disk as it is. There's no need to enter the comments after the semicolon, as we've included them just to explain the operations, although you can if you wish. We'd recommend entering the code from the keyboard rather than using the existing "file" to get some experience in using the Editor and Assembler.

```
100 ;LOAD THE 8-BIT REGISTERS
110 START      LD      A,55      ;load A with 55
120            LD      B,55H     ;load B with 85
130            LD      C,0       ;load C with 0
140            LD      D,255     ;load D with 255
150            LD      E,OFFH    ;load E with 255
160            LD      H,176     ;load H with ??
170            LD      L,128     ;load L with ??
180            END              ;end this section
```

4 Loading Registers

Ok, are you ready to assemble? Assemble by entering A and check for errors. If you have a “clean” assembly, you should see the display shown in Figure LESS4-1, after doing a P#. If you have errors, go back and correct the individual lines by using the Edit commands and List until you have the same lines of code. Then reassemble until you have no errors.

```
PC  AH AB      S Z H P N C B C D E H L IX IY SP
FFFF FF 11111111 1 1 1 1 1 1 FF FF FF FF FF FF FFFF FFFF FFFF
.....
STATUS LOCN CONTENTS LINE LABEL OPCODE OPERAND COMMENT
      B2C4      00099 :.....LESSON 4.....
      B2C4      00100 :LOAD THE 8-BIT REGISTERS
      B2C4 3E 37      00110 START LD A,55
      B2C6 06 55      00120 LD B,55H
      B2C8 0E 00      00130 LD C,0
.....
LOCN 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
.....
MODE=EDIT SPEED=9000 FREE MEMORY=19452 BP @ FFFF,FFFF,FFFF,FFFF
COMMAND=
```

NOTE: "LOCN" VALUES MAY
VARY FROM THOSE
SHOWN. THERE MAY
BE OTHER MINOR
FORMAT DIFFERENCES

Figure LESS4-1. Lesson 4 Assembly

Before you “execute” the program, look at the register display at the top of the screen. Let’s review what each of the labels stands for.

The PC is the Program Counter. It contains the location of the machine-language instruction currently being executed. The PC is 16 bits long, and the four hexadecimal digits represent those 16 bits in a “shorthand” hex notation.

The AH label stands for “A register in Hex.” Since the A register is 8 bits long, there are two hexadecimal digits to represent what it is holding. The AB label stands for “A register in Binary.” This is the binary representation of the A register in 8 bits, which will be 0s or 1s.

The next 6 labels, S, Z, H, P, N, and C are the “flags” of the F register. We won’t be using these in this lesson, but you might keep your eye on them to see how they change throughout some of the following programs. They generally change to reflect the results of arithmetic and other processing.

The next labels are for the B, C, D, E, H, and L registers. As these are 8-bit registers they have two hexadecimal digits each. The last three labels are IX, IY, and SP; we won’t be using these registers in this lesson.

Now set the speed of execution by

ZS O

This is a “slow speed” setting of about 1 instruction per three seconds that will allow you to watch the registers change.

Ready to execute? Enter

ZX

to execute from the START of the program.

You’ll see the PC register change as each instruction executes. You’ll also see each register change as it is “loaded” with a binary value. After the program has stopped, you’ll see the display shown in Figure LESS4-2.

THIS VALUE WILL BE DIFFERENT FROM THAT SHOWN

THESE VALUES WILL BE DIFFERENT FROM THOSE SHOWN

PC	AH	AB		S	Z	H	P	N	C	B	C	D	E	H	L	IX	IY	SP
FFFF	37	00110111		1	1	1	1	1	1	55	00	FF	FF	B0	80	FFFF	FFFF	FFFF
.....																		
STATUS	LOCN	CONTENTS		LINE	LABEL	OPCODE	OPERAND		COMMENT									
.....																		
(THIS AREA AREA WILL HAVE TEXT IN IT)																		
.....																		
LOCN	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F		
.....																		
MODE=EDIT SPEED=9000 FREE MEMORY=21360 BP @ FFFF,FFFF,FFFF,FFFF																		
COMMAND=																		

Figure LESS4-2. Lesson 4 Register Display

Each of the 7 registers has been loaded with a data value. Look at the A register, for example. In the source code line, we had an “LD A,55” instruction, standing for “Load A with 55.” If you look at the AB display, you’ll see 00110111, which is the binary equivalent for decimal 55. The hexadecimal version of this (37H) is shown in the AH display.

What about the B register? We loaded it with a 55H. The “H” is a special hexadecimal suffix which says that the number for the load is in hexadecimal in the source code line. Sure enough, if we look at the B display, we’ll see a 55H.

Look at the display for C, D, E, H, and L. Do they correspond to the values that we specified in the source lines?

By the way, what is the largest number that can be held in 8 bits? In other words, what is the largest number that can be loaded into an 8-bit general register? We’ve loaded the D and E registers with this value, which is a decimal 255, or a hexadecimal 0FFH.

We’ll leave the conversion of the decimal values 176 and 128 up to you. These values are loaded into the H and L registers, respectively, and you can find the equivalent hex values by looking at the contents of H and L from the display.

Immediate Loads

This type of instruction used an “addressing” mode called “immediate” addressing. In this type of addressing the data for the instruction is contained in the instruction itself. Look at the assembly listing for the LD H,176, for example.

In this case we wanted to load 176 into the H register. The value of 176 decimal is (as you’ve figured out) 0B0H, and if you look in the machine-language data for the LD H,176, you’ll see a 26 byte, followed by an B0. In fact, all of these instructions have the “immediate” 8-bit data to be loaded in their second byte.

Anytime that you see an “LD” in the source line, followed by the “mnemonic” of an 8-bit general-purpose register, followed by a value in decimal or hexadecimal **without parentheses** around the value, the LD will be an “immediate 8-bit load” that loads data into the register as we’ve seen.

Loading 16-Bit Registers

How about loading 16-bit registers with immediate data? Is the procedure the same? The LD for 16-bit registers is very similar. Again, we’ve got an LD mnemonic, followed by the register mnemonic, followed by the data value without parentheses.

4 Loading Registers

To see an example of this, delete lines 100 through 180, and then enter the following code, or use what is in the source file.

```
190 ; 16-BIT LOADS
200 NEXT      LD      BC,1000      ;load BC with 1000
210          LD      DE,OFFFHH     ;load DE with maximum
220          LD      HL,18H        ;load HL with 18H
230          END                  ;end
```

Assemble the source code, check it over to see that it corresponds to the source code above, and assemble it. If you have no assembly errors, execute the program by doing a

ZX

After you've executed the program, look at the register display. As you might have guessed, the three instructions above load the BC register "pair" with a value of decimal 1000, the DE register pair with hexadecimal OFFFHH, and the HL register pair with 18H. How does this appear in the display?

Look at the B and C registers first. When loaded together, they act as a single 16-bit register with B being the "upper" 8 bits and C being the "lower" 8 bits. B is called the "most significant byte" and C is called the "least significant byte."

Let's see, 1000 in binary is 000000111101000, or in hexadecimal, 03E8H. If we look at B we can see that it contains the most significant byte of 03H, while C contains the least significant byte of E8H. Taken together, the value is 03E8H, what it should be for decimal 1000.

Now look at the D and E registers. Here the load was of OFFFHH. What is OFFFHH in decimal? It turns out that OFFFHH in hex is decimal 65,535, the largest number that can be held in 16 bits, and this value is in D and E, with each holding FFH.

One more for practice. HL was loaded with 18H. What should the result be in H and L? A value of 18H in 16 bits is 0000000000011000, with the most significant byte equal to 00H, and the least significant byte equal to 18H. H, therefore, holds 00H, while L holds 18H. Note that the number is "padded" with zeroes to the left to make up 16 bits.

Now look at the instructions for the 16-bit LDs. The machine-language code for the instructions is on the left of the assembly display, or on the left of a line printer listing of the assembly. Since the data to be loaded into the register pairs is 16-bits long, we'd expect to see 16 bits or 2 bytes of immediate data, and that's what we see for the LDs.

The LD BC,1000, for example, consists of a byte of 01 for the "operation code" of the LD and two bytes of E8H and 03H for the data. But wait, there's something wrong here! The data appears to be reversed. It looks like the most significant byte follows the least significant byte! Why?

It turns out that this is the standard way the Z-80 represents 16 bit data — least significant byte followed by most significant byte. You'll see this form of representation in all 16-bit data — immediate data, addresses, and other data, so get used to it! In the LD HL,18H case, for example, we see 18H, followed by 00H. We'll remind you of this representation from time to time, lest you forget.

Transferring Data Between CPU Registers

We've seen how to load 8- and 16-bit registers with immediate data, but what about "moving" data between cpu registers? This is done all the time in assembly-language programs. Often we want to move data from the A register into another 8-bit register and back again.

Again we can use the LD instruction. This time, however, we won't be loading immediate data but copying the contents of one register into another.

Delete source lines 190 through 230, and enter the following program, or use the existing source lines from the program in RAM.

```

240 : LOAD BETWEEN REGISTERS
250 LAST      LD      A,0BH      ;immediate load of 0BH
260          LD      B,A        ;now in B
270          LD      C,0DH      ;immediate load of 0DH
280          LD      D,C        ;now in D
290          LD      C,0CH      ;immediate load of 0CH
300          END                ;end

```

This short program shows you how the registers can transfer data to one another. Enter it, edit it until it assembles properly, and execute it at slow speed. At the end, B should contain 0BH, C should contain 0CH, D should contain 0DH, and A should contain 0BH.

Any 8-bit register can be loaded by any other 8-bit register simply by using the LD D,S form of the load instruction. In this instruction, the proper assembly-language form uses "D" for the "destination" register and "S" for the "source" register. The source register always loads into the destination register. At the end of the load, the source register remains unchanged. Try entering your own source code and moving data between other registers, and you'll see how it works.

What about moving data between 16-bit registers? Can we do a LD HL,BC, for example, to load the HL register pair with the contents of the BC register pair? No, these instructions are not implemented in the Z-80. You'll have to do a

```

LD      H,B      ;load H with B
LD      L,C      ;load L with C

```

to accomplish the load, or use the "stack" as we'll show you a little further on.

It's also possible to perform loads of 16-bit registers to HL, IX, and IY by first loading the register with 0 ("clearing" the register) and then adding another register pair to HL, IX, or IY; we'll discuss these special adds in another lesson.

To Sum It All Up

To recap what we've learned in this lesson:

- You can load any general-purpose 8-bit register (A, B, C, D, E, H, or L) with an 8-bit immediate value by an "immediate" load
- You can load any 16-bit general-purpose register pair with a 16-bit immediate value by an "immediate" load
- Sixteen-bit data in the Z-80 is always ordered least significant byte, followed by most significant byte
- You can load any 8-bit general-purpose register with any other 8-bit general-purpose register by an LD of the form LD D,S, where S is "source" register and "D" is "destination" register
- You can't load a 16-bit register pair with another 16-bit register pair directly, but you can do it indirectly by loads on individual 8-bit registers or other techniques

For Further Study

LD 8-bit immediate instructions (see Appendix V)
 LD 16-bit immediate instructions
 LD D,S instructions
 ADD HL, ADD IX, ADD IY instructions



Lesson 5

Loading and Storing Between CPU Registers and Memory

Load LESS5 from disk.

In the last lesson we discussed how data could be loaded and moved between cpu registers. In this lesson we'll see how data can be moved between the cpu registers and memory.

When data is moved from memory to a cpu register, a "load" is performed. When data goes the other way, from a cpu register to memory, it is said to be "stored." Here again, the meaning of these terms is historical and dates back to early computers.

In either the load or store case, the general instruction type involved is an "LD." Other microprocessors use an "ST," but the Z-80 uses an "LD" rather than a store mnemonic.

In general, you can tell which direction the transfer will be by the same "source," "destination" format that we saw in the LD of cpu registers. Parentheses are used to denote that the operation is to and from a memory location rather than an immediate operand.

The destination is always first, followed by the source in the format LD D,S. As an example,

LD **(8000H),A** ;store A to RAM

stores the contents of the A register (the "source") to memory location 8000H (the "destination"). On the other hand:

LD **A,(8000H)** ;load A with RAM value

loads the A register with the contents of memory location 8000H.

Eight-Bit Direct Loads and Stores to Memory

Enter the source program shown below, or use the source from the Lesson File.

```
100 ; 8-BIT DIRECT LOADS AND STORES
110 START       LD        A,33H               ;load A with 33H
120               LD        (0B000H),A       ;store in B000H
130               LD        A,0               ;clear A
140               LD        A,(0B000H)       ;load A with B000H
150               END
```

Check the source lines by reviewing them with the Edit commands. Again, you don't need to enter comments unless you wish. When you have a good set of source statements, assemble the program by the A command and check for errors. If you have no errors, go on to the next part, but if you have errors, re-edit and reassemble.

Before executing this short program, "trace" memory locations B000H through B01FH by entering

ZT B000

After the ZT, you should see the B000H area displayed in the experiment area.

Now execute the program by doing

ZS 0
ZX

5 Loading and Storing Between CPU Registers and Memory

The program will first do an immediate load of 33H into the A register. It will store the contents of A into RAM memory location B000H. The A register will then be cleared by an immediate load of 0. Finally, a load of location B000H will be done. This will load the 33H at B000H back into the A register.

This type of load and store uses **direct addressing**. In direct addressing the RAM address for the load or store is directly specified in the instruction itself. Look at the LD (0B000H),A or LD A,(0B000H) instruction. In the instruction, you'll see the address for the load or store in standard Z-80 address format in the second and third bytes — 00B0H.

Direct loads and stores are the easiest way to store 8 bits from the A register into a single memory byte or to load a single memory byte into the A register. What about the other 8-bit general purpose registers? Can we do direct loads and stores on them, also?

Unfortunately, there are no instructions to do a direct load or store of the other 8-bit registers. There are no LD B,(0B000H) instructions, for example. We have to use other means to move data between these registers and memory.

Eight-Bit Indirect Loads and Stores

One way to load and store the 8-bit registers is by **register indirect addressing**. In this type of addressing, the HL register, BC register, or DE register is used as a “pointer” register, pointing to the RAM memory location for the load or store, as shown in Figure LESS5-1.

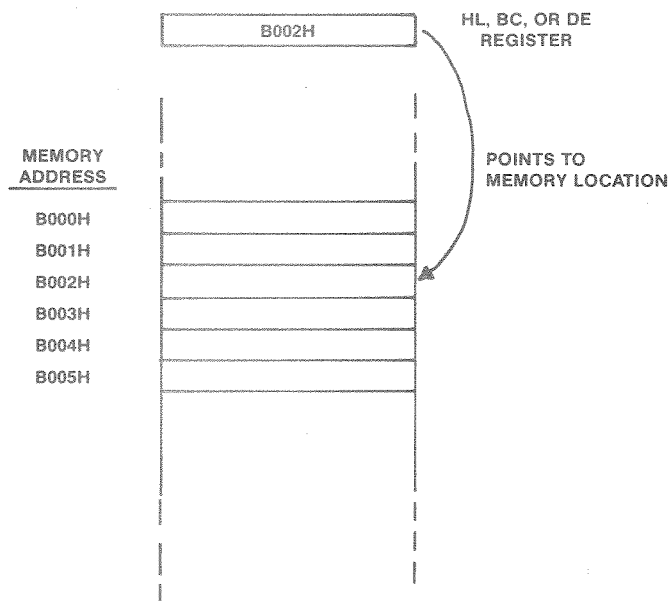


Figure LESS5-1. Pointer Registers

Loading and Storing Between CPU Registers and Memory 5

To see how this works, delete lines 100 through 150, and enter the program below, or simply use the existing source code from the Lesson File.

```
160 ; 8-BIT INDIRECT LOADS AND STORES
170 NEXT      LD      HL,0B000H      ;pointer to B000H
180          LD      A,0AH          ;load A into A
190          LD      B,0BH          ;load B into B
200          LD      C,0CH          ;load C into C
210          LD      D,0DH          ;load D into D
220          LD      E,0EH          ;load E into E
230          LD      (HL),A         ;store A
240          INC     HL             ;now B001
250          LD      (HL),B         ;store B
260          INC     HL             ;now B002
270          LD      (HL),C         ;store C
280          INC     HL             ;now B003
290          LD      (HL),D         ;store D
300          INC     HL             ;now B004
310          LD      (HL),E         ;store E
320          LD      A,(HL)         ;load A with B004
330          LD      B,(HL)         ;load B
340          LD      C,(HL)         ;load C
350          LD      D,(HL)         ;load D
360          LD      E,(HL)         ;load E
370          END
```

After you've checked the source lines and assembled without any errors, execute the program by

ZX

What did you see? First of all, the HL register is set up as an "indirect pointer" register by loading it with B000H in an immediate load. Next, the A, B, C, D, and E registers are loaded with the values of 0AH, 0BH, 0CH, 0DH, and 0EH, respectively.

Now the A register is stored "register indirect" by the LD (HL),A instruction. What would you think that this instruction does? We know that instructions generally use the format "destination"/"source." In this case it looks like the source is A, and the destination is (HL).

Just as the parentheses around (0B000H) specified a RAM memory address for a direct addressing instruction, the (HL) specifies a memory address. In this case the memory address is the "pointer" address contained in the HL register. As this was loaded with B000H initially, the LD (HL),A instruction is logically equivalent to an LD (0B000H),A.

In the next few instructions we're storing the other registers, B, C, D, and E, to memory also, using the HL register pair as a pointer. Between each store, the HL register is "incremented" by the instruction INC HL. Look at how HL changes as you execute the program. It changes by 1 for each INC. The increment and its inverse, the DEC HL, are common ways to modify HL when it is used in this register indirect addressing mode.

After we've stored A, B, C, D, and E into memory locations B000H through B004H, HL points to B004, which contains 0EH. As a grand finale, we're loading the registers using HL as a pointer. The 5 loads work the same way, except that the "source" is now (HL) and the destination is the register.

Using HL as a pointer is a common way to do 8-bit stores and loads for A through E. It works best when the area in memory is a "table" of data, with "contiguous" locations, rather than memory locations scattered around memory in "random" locations. Why?

5 Loading and Storing Between CPU Registers and Memory

If we used HL as an indirect pointer for random data, we'd have to load it with new address values for each load or store. For example, if we wanted to store A, C, and E in three different memory locations we might have something like

LD	HL,LOCA	;first location
LD	(HL),A	;store A
LD	HL,LOCB	;second location
LD	(HL),C	;store C
LD	HL,LOCC	;third location
LD	(HL),E	;store E

Here, we've used "LOCA," "LOCB," and "LOCC" to represent absolute addresses that might have been B000H, B023H, and B100H, or similar addresses. You can see that this "random addressing" using the HL as a pointer is not too efficient, compared to the one-byte INC HL or DEC HL.

HL is the main register used in this indirect addressing mode. As a matter of fact, the H in HL stands for "High" and the L in HL stands for "Low" — the high and low being the high and low address bytes. And no, I'm not just making this up either...

The Z-80 is an relative of the 8080 microprocessor and contains the instructions of the 8080 as a "subset." The 8080, in turn, is a relative of the 8008, an even more basic microprocessor. Originally (in the 8008) the HL register was the only register that could be used for indirect addressing. In the 8080, however, register pairs BC and DE were also given their own indirect capability, but only for the A register. To see how these instructions work, delete lines 160 through 370, and enter the source code below:

```
380 ; 8-BIT INDIRECT ADDRESSING USING BC AND DE
390 ANUDR      LD      BC,OB000H      ;set up address
400           LD      DE,OB008H      ;set up address
410           LD      A,OAAH          ;load 170
420           LD      (BC),A          ;store into B000H
430           LD      (DE),A          ;store into B008H
440           END                    ;end
```

This program doesn't do a great deal, but it does show you that BC and DE are used in exactly the same way as HL in the indirect addressing mode, but only for the A register.

Assemble and execute the program.

Wow! Too many combinations of things, eh? You can use HL as an indirect pointer for any register, but you can only use BC or DE as an indirect pointer for A, but only if the phase of the moon is right... That's the way it is in Z-80 assembly language; you must learn which registers use which addressing modes. Although the Z-80 instruction set is not as "generic" as others, it is very powerful in spite of the special cases!

Sixteen-Bit Loads and Stores

All of the loads and stores above have been 8-bit loads and stores. What about moving 16 bits of data from cpu registers to memory? This is possible by using the 16-bit direct addressing mode, where the instruction contains the memory address to be used. No "register indirect" operations are possible.

BC, DE, HL, SP, IX, and IY can be stored in this manner. We will cover only BC, DE, and HL and leave the other registers for later discussion for your own study.

To see how these direct loads and stores are done, delete lines 380 through 440 and enter this next set of code:

450 ;16-BIT DIRECT LOADS AND STORES

```

460 STILAN      LD      BC,1234H      ;load sample data
470             LD      DE,0ABCDH     ;load sample data
480             LD      HL,1000       ;load sample data
490             LD      (0B000H),BC   ;store
500             LD      (0B002H),DE   ;store
510             LD      (0B004H),HL   ;store
520             LD      BC,(0B004H)   ;load BC with 1000
530             LD      DE,(0B004H)   ;load DE with 1000
540             END

```

Edit and assemble this program as you have been doing. When you have an error-free assembly, execute the program after first “tracing” the B000H area:

```

ZT B000
ZX

```

What do we have during execution? You’d expect the contents of BC, DE, and HL to be stored in the experiment area starting at B000H on. Let’s see, B000H contains 34H and B001H contains 12H. If we reverse these, we have 1234H, which is the contents of BC before the store. Aha! Note that again the Z-80 stored 16-bit data in “reverse format,” putting the least significant byte first followed by the most significant byte. Also, we used 2 memory locations, B000H and B001H. The instruction itself points to the **first** byte, even though 2 bytes will be used.

Now look at B002H and B003H. You’d expect these locations to contain the original contents of DE, and sure enough, CDH is in B002H and ABH is in B003H. The next two locations contain E8H and 03H, 1000 decimal in reverse order.

The last instructions load BC and DE with locations B004H and B005H, the decimal 1000 originally in the HL register. Note that, again, the **first** byte of the data is specified in the instruction, but that 2 bytes are loaded from memory.

IX and IY Used as Indirect Registers

Up to this point we’ve avoided mentioning the IX and IY index registers. The IX and IY index registers are used in indexed addressing mode, a way of easily accessing sequential (rather than random) data that is grouped together in memory.

The IX and IY can be used as indirect pointers also, in the same fashion as HL. Any time that an 8-bit load or store can be done with HL, you can do it with IX or IY; from our work above, that means that IX and IY can be used with A, B, C, D, E, H, and L to load or store these 8-bit registers.

You might experiment with IX and IY using these instruction formats:

```

LD      IX,0B000H      ;load address
LD      (IX),A          ;store A

```

This doesn’t use the full capabilities of IX and IY, but does show you how they can be used in this “degraded” mode, a common way of using IX and IY in many programs.

To Sum It All Up

To review what we’ve learned in this lesson:

- A load means that data will be loaded into a cpu register from memory
- A store means that data will be stored into memory from a cpu register
- Both the load and store use the LD instruction mnemonic

5 Loading and Storing Between CPU Registers and Memory

- The A register can be loaded and stored directly by using the direct addressing mode where the instruction contains the memory address to be used
- Registers A, B, C, D, and E can be stored by using HL as an indirect pointer in the register indirect addressing mode
- The BC and DE registers can be used as indirect pointers, but only for loading and storing A
- Register pairs BC, DE, and HL (also SP, IX, IY) can be loaded or stored directly to or from memory
- The IX and IY registers can be used as indirect register pointers similarly to HL

For Further Study

Use of IX and IY as indirect pointers — private study

Lesson 6

Adding and Subtracting 8- and 16-Bit Numbers

Load LESS6 from disk.

We haven't done much as far as using the Z-80 as a **computer**, in the common use of the word. All we've done up to this point is to "shuffle" data around between cpu registers and memory, an important task, but not all that exciting. In this lesson we'll get down to doing actual arithmetic with the Z-80 — 8-bit and 16-bit additions. That may not sound like much in these days of \$10 calculators that perform square roots and more, but it's the foundation of much Z-80 assembly-language code.

Eight-Bit Adds

The Z-80 has one basic add: An 8-bit operand from a cpu register or from memory is added to the contents of the A register. The result is then put back into the A register. Let's see how this works. Enter the source code below, or use the existing source code from the Lesson File:

```
100 ;8-BIT ADDS
110 START      LD      A,100      ;100 to A
120            LD      B,150      ;150 to B
130            ADD     A,B        ;add A and B
140            LD      (B000H),A   ;save result
150            LD      A,37        ;37 to A
160            ADD     A,A        ;add A and A
170            LD      (B001H),A   ;save result
180            ADD     A,10        ;add 10 to A
190            LD      HL,B001H    ;point to B001H
200            ADD     A,(HL)      ;add A and B001H
210            END                ;end
```

Edit the source code until it is identical to the source code above, except for the optional comments. Assemble the code to get an error-free assembly.

Before you execute the code, use a slow speed and "trace" the B000H area by entering:

```
ZS 0
ZT B000
```

Here's what should happen on execution: You should see A loaded with 100 and B loaded with 150 by the two "immediate" loads. Now the ADD A,B should take the contents of B and add it to A. The result will be stored in memory location B000H. A is now loaded with 37. Now A should be added to itself by the ADD A,A, and the result should be stored in location B001H. Next, 10 should be added to the A register. Next, the HL register pair should be loaded with the value B001H. HL will be used as a register indirect pointer to location B001H. The last operation adds the contents of A with the location pointed to by HL, location B001H.

Execute the program now, and try to follow the operations.

The result of the first add, ADD A,B, should have added the 150 in B to the 100 in A. A value of 150 decimal is 96H, while a value of 100 is 64H. (You can check these values by looking at the immediate loads of A and B.) The result in location B000H should be 250, or a value of FAH, and we can see from the trace that this is the value put in location B000H.

The second add adds A to itself and puts the result in location B001H. This is a perfectly legitimate

6 Adding and Subtracting 8- and 16-Bit Numbers

operation. (As a matter of fact, you can even do an LD A,A, although it's rather meaningless.) We started out with 37 decimal in A (25H), and we should wind up with 74 decimal in location B001H. Our trace indicates that location B001H is hexadecimal 4A, which is correct.

The next add adds 10 to the contents of A by "immediate" addressing. The total of 74 now becomes 84 decimal.

Our last operation adds location B001H, containing the 4AH, to the contents of the A register, a 54H. The result should be 74+84 or 158. Looking at the A register after the execution, we see that it contains 9EH, which is equivalent to 158 decimal.

In the program above, you can see the types of addressing modes that we can use with 8-bit adds to A. The first is register-to-register — another cpu register added to A, including A itself.

The next addressing mode is the "immediate" type of addressing, which adds an immediate value from the instruction itself to the contents of the A register.

The next is register indirect, adding a memory location pointed to by HL with the contents of A.

We've left off another type of addressing mode which is also possible, indexed addressing, using the contents of the IX and IY registers; we'll discuss indexing in another chapter.

This collection of addressing modes — register to register, immediate, HL indirect addressing, and indexed addressing, is typical for all instructions that use the A register. The same addressing modes would be available for a subtract, an OR operation, or an increment of A. As each addressing mode counts for a separate instruction type, you can see where some of the many separate instructions of the Z-80 come from. Many instructions are just the same instruction with a different addressing mode!

Sixteen-Bit Adds

All 8-bit adds use the A register. What would you expect 16-bit adds to use? Right, the HL register, which is a "16-bit" accumulator.

The HL register was used to perform 16-bit adds in the 8080 microprocessor; the Z-80, however, kept the original ADD and also added ADDs to two other registers, the IX and IY registers. So we can say that there are really three "16-bit" accumulators — HL, IX, and IY, at least for adds.

To see how these registers add two numbers, delete lines 100 through 210, and enter the source lines below:

220 ; 16-BIT ADDS USING HL, IX, IY

230 NEXT	LD	BC,1000	;1000 decimal
240	LD	DE,500	;500 decimal
250	LD	IX,250	;250 decimal
260	LD	IY,100	;100 decimal
270	ADD	IX,BC	;250+1000
280	ADD	IX,IX	;1250+1250
290	ADD	IY,BC	;100+1000
300	LD	HL,2000	;2000 to HL
310	ADD	HL,DE	;2000+500
320	END		;end

When you have an error-free assembly, execute the code by entering

ZX

At the end of this program, the results will be in the HL, IX, and IY registers. We added the contents of the BC register pair to IX and then added IX to itself. The result should be 2500, which is 09C4H.

Next, we added the contents of the BC register pair to IY, 1000+100. The value 1100 decimal is 044CH in hexadecimal, and this is what we should see in IY at the end of the program.

The last 16-bit add added the contents of the DE register pair to the HL register, 500+2000, and there should be a 09C4H in HL after the program stops.

How did these 16-bit adds differ from the 8-bit adds? For one thing, of course, twice the “width” of data was added, which means that sums up to 65,535 can be handled, rather than only 0 through 255. For another thing, the 16-bit adds are rather limited. The operand to be added to the contents of HL, IX, and IY must already be in another register! No sophisticated addressing modes are available for the 16-bit adds.

Eight-Bit Subtracts

The 8-bit subtract is very much like an 8-bit add. It operates only on data in the A register, and no other cpu registers, and has the same addressing modes. In the subtract, an 8-bit operand from another cpu register or from memory is subtracted from the contents of the A register, with the result going into the A register — it’s identical to the ADD except for the basic operation and the fact that you don’t need an “A” to denote the destination.

To show you how the subtract works, delete lines 220 through 320, and enter the following subtract example, or use the source code from the Lesson File.

330 ; 8-BIT SUBTRACTS			
340 NEXT1	SUB	A	;subtract A from A
350	LD	(B000H),A	;store result
360	LD	A,146	;load with 146
370	SUB	13	;146-13
380	LD	(B001H),A	;store result
400	LD	A,100	;load with 100
410	LD	HL,B001H	;set up address pointer
420	SUB	(HL)	;subtract 133 from 100
430	LD	(B002H),A	;store result
440	END		;end

Check the source lines before assembling and then assemble to get an error-free assembly. Trace the B000H area as before. Execute by

ZX

Now for the results.

What would you expect the first subtract of SUB A to be? This instruction specifies that the contents of the A register will be subtracted from the contents of the A register. Since any number subtracted from itself is 0, you’d expect the result in B000H to be 0, and it is. This is one way of clearing the A register. Another way, of course, is by doing an LD A,0. Which is better?

Ordinarily, this is a question that wouldn’t be too significant, but since A is cleared many times in assembly-language programming we might want to consider it here. The LD A,0 is a 2-byte instruction, while the SUB A is a 1-byte instruction. The LD A,0 occupies twice as much memory and operates almost twice as slowly as the SUB A. It’s better programming practice, then, to use a SUB A to clear A, although we’ll see another clear of A, the XOR A, in another lesson.

The next subtract subtracted an immediate value of 13 from the 146 loaded into the A register in the previous instruction. The result of 133, equivalent to hexadecimal 85H, is in location B001H after the program end.

The next subtract is an illustration of a register indirect using the HL register. The A register is first

6 Adding and Subtracting 8- and 16-Bit Numbers

loaded with 100 by an immediate load. Next, the HL register is loaded so that it points to location B001H. A subtract using the register indirect is then done, effectively subtracting 133 (in B001H) from the 100 in the A register. What will be the result of this subtract? We expect to get a -33, but get the result DFH in location B002H instead. To answer that question we're going to have to look at a number representation called "two's complement."

Two's Complement Numbers

Up to this time we've been working with "absolute numbers" held in 8 to 16 bits. The binary numbers we've considered have always been positive, integer values. This makes sense in many cases. Take the HL register values, for example. The HL was originally used to represent a memory address from 0 through 65,535, and there is no such thing as a "negative address."

However, we would like to be able to represent both positive and negative numbers. (I need some way to handle balances in my checking account...) How is it done?

The scheme that the Z-80 and almost all other microprocessors or computers use is called "two's complement." The format of two's complement numbers is shown in Figure LESS6-1.

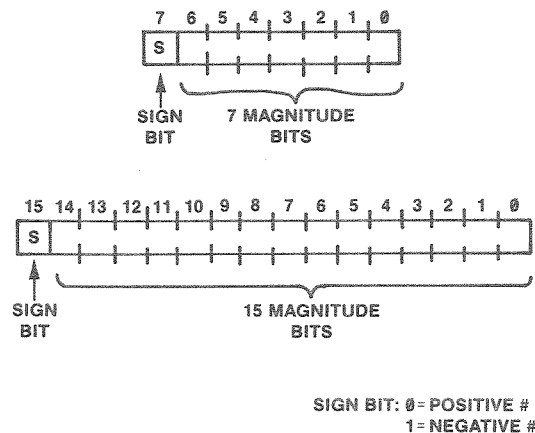


Figure LESS6-1. Two's Complement Numbers

The most significant bit of an 8 or 16-bit number is designated as a "sign bit." The remaining bits are "magnitude bits."

If the sign bit is a 0, well and good. The remaining 7 or 15 bits represent the "magnitude" of a positive number. In 8-bit values you can therefore have positive numbers from 0 0000000 (0) through 0 1111111 (+127).

If the sign bit is a 1, however, the number represents a negative value. In that case, change all the 0s to 1s, change all the 1s to 0s, and add 1. Why? A purely mechanical process that gives you the magnitude of the negative value.

Let's take the last subtract and show you how it's done. The result was DFH, or binary 11011111. The most significant bit was a 1, so the number is a negative number. Changing all 1s to 0s and all 0s to 1s gives us 00100000. Adding 1 gives us 00100001. 00100001 is 33 decimal, and therefore the negative number is -33, the result we should expect to get by subtracting 133 from 100.

Negative values of -1 (11111111) through -128 (10000000) can be held in 8 bits. (Note that although we can hold a negative number of -128, the maximum positive number that can be held is +127.)

Two's complement works exactly the same in 16 bits. The same actions are taken to convert. Look at the sign bit first, and if it is a 0, the number is a positive number from 0 (0000000000000000) through +32,767 (0111111111111111). If the sign bit is a 1, the number is a negative number from -1 (1111111111111111) through -32,768 (1000000000000000).

Note that the two's complement number ranges are the same as BASIC integer values. All basic integer values are in reality two's complement 16-bit values!

Why are two's complement numbers used? So that we can easily do adds and subtracts with the ADD and SUB instructions in the Z-80. We don't have to laboriously check each number to see if it's positive or negative, we just go ahead and do the add or subtract — the number will be adjusted accordingly. We'll discuss this more in Lesson 10, along with other arithmetic operations.

To Sum It All Up

To review what we've learned in this lesson:

- Eight-bit adds use the A register as the destination and either another 8-bit cpu register (A, B, C, D, E, H, L) or an 8-bit memory operand for the add
- Eight-bit adds can use register-to-register, immediate, register indirect, or indexed addressing modes
- Sixteen-bit adds use either HL, IX, or IY as a "16-bit accumulator"
- Sixteen-bit adds add the operand from another 16-bit register to the contents of HL, IX, or IY; no other addressing modes can be used
- Eight-bit subtracts are virtually identical to 8-bit adds as far as the handling of operands and addressing modes
- Two's complement numbers express both positive and negative integer values. If the sign bit is a 0, then the remainder of the number determines the positive magnitude, otherwise a simple conversion must be done to find the negative number

For Further Study

Two's complement numbers — Appendix VI



1
2

Lesson 7

Adds, Subtracts, and Flags

Load LESS7 from disk.

In this lesson we're going to investigate the Flags of the Z-80. The Flags are a collection of 8 bits, as shown in Figure LESS7-1. There are really only six flags, as two of the 8 bits in the F "register" are unused.

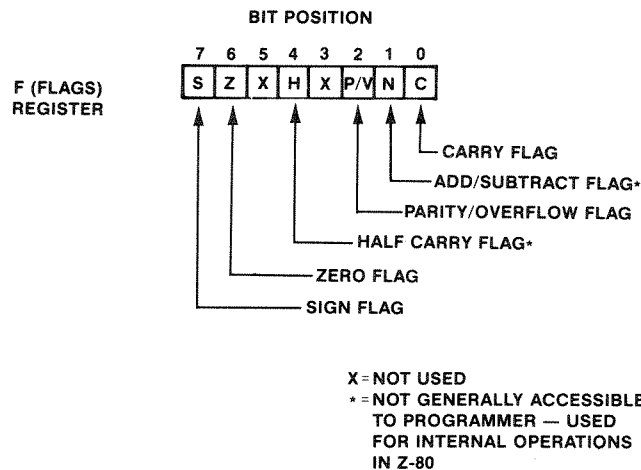


Figure LESS7-1. Flags

The Flags are logically grouped together with the A register, but not in the same way that B is grouped with C, D is grouped with E, and H is grouped with L. The alignment of the F register with A is only a convenience in saving this pseudo-register pair of AF in the "stack," an operation we'll discuss in another chapter.

The Six Flags (and these are not necessarily over Texas) reflect the results of arithmetic and other instructions, as we'll see. The flags can alter the program flow, as they can be tested by "jump" instructions, which cause jumps to different program code on the settings of the flags, which in turn are set by prior arithmetic.

Here's an example:

```
SUB      B      ;A — B
JP      Z,EQUAL ;go if A=B
                ;not equals here
```

The first instruction above subtracted the B register from the A register. The subtract instruction affects most flags. If the result is zero, the Z flag is set as one of the last actions in the subtract. The JP instruction is a "conditional jump" that jumps to location EQUAL if the Z flag is set but otherwise doesn't "take the jump" and "falls through" to the next instruction in sequence. The mnemonic "JP Z," means "Jump if Z flag set"; set always means a 1 condition, while reset means a 0 condition.

More on Adds and Subtracts — the Z Flag

Let's look at some of the flag actions during 8-bit adds and subtracts, as we're already experts on these! Enter the code below, or use the existing source code from the Lesson file.

7 Adds, Subtracts, and Flags

100 ; ARITHMETIC FLAG ACTIONS

110 START	LD	A,33	;A operand
120	LD	B,33	;register operand
130	SUB	B	;33-33=0
140	LD	A,32	;A operand
150	SUB	B	;32-33
160	LD	A,34	;A operand
170	SUB	B	;34-33
180	END		;end

Check the source code to see that it matches the listing above. Assemble the code so that you get an error-free assembly.

Now set the speed of the Interpreter to the slowest setting by entering

ZS O

This will let you see the Flag settings as the instructions are executed at slow speed. Keep your eye on the Z flag in the register display and observe how it changes in running the program above.

Ok, ready? Execute the program by

ZX

Did you catch the Z flag settings? The first subtract subtracted 33 from 33. The result of this was 0 in the A register, so the Z flag should have been set (1) to indicate a Zero condition. The next subtract subtracted 33 from 32. The result of this is -1, or FFH, definitely a “non-zero” condition, and the Z flag should have been reset (0) to indicate non-zero. The third subtract subtracted 33 from 34; this is also a non-zero result, and the Z flag should have remained reset at 0 for “NZ.”

If you didn’t catch the flags the first time, run the program again and try to observe the Z flag.

The Z flag, therefore, is set or reset according to the results of the subtract. It is also set for adds and other instructions. Here’s an important point, though: The Z flag and other flags are unaffected by certain instructions! All LD instructions, for example, leave the flags unchanged from what they were in the previous instruction.

In most cases you’ll be using the flag setting directly after the add, subtract, or other processing in a conditional jump, so there won’t be intervening instructions that could affect the flags. In other cases the “test” of the flags by a conditional jump may be several instructions away. It’s important, therefore, to know which instructions affect the flags and which ones do not. You can find this information in Appendix V, where all flags are listed for every instruction type.

The S(ign) Flag

Another flag in the Flags register is the S flag, standing for “Sign.” The S flag is set (1) if the result of the operation is negative and reset (0) if the result of the operation is positive. Here again, the S flag is not set for all instructions but is set for adds, subtracts, and other arithmetic and logical operations.

Execute the program above again, but this time watch the settings of the S flag.

Got them? You should have seen the S reset (0) as the first subtract of 33-33 was done. This indicates that the result is a positive number. (Zero is always a positive number in the Z-80. If we apply the rules of two’s complement “notation” and look at the sign bit, we see a positive number with a magnitude of 0000000.)

The next subtract was 32-33. The result here should have been a negative value of -1. The S bit here was set (1) after the subtract to indicate that the result was negative.

The last subtract was 34-33 for a result of 1, a positive number again. The S flag was reset (0) after this subtract.

Here's one of those interesting questions (that usually occur at 2:00 a.m.) How does the Z-80 know whether we are subtracting in two's complement or in absolute form? In other words, we might be working with absolute numbers in A from 00000000 through 11111111 (255) instead of two's complement numbers of -128 through +127. The answer is: The Z-80 **doesn't** know! It blithely sets the S flag as if it were two's complement operations. However, we know, and if we are operating in absolute numbers up to 255, we ignore the S flag.

The Compare

There's an important variation of the SUB instruction that we've ignored up to this point. This is the CP, or Compare instruction. The compare works exactly like the SUB, except that it does not put the result back into the A register. It simply drops the result into the "bit bucket" on the floor behind the TRS-80.

What the compare **does** do, however, is to set the Z, S, and other flags. We can use the compare to test one operand against another without destroying the contents of A register.

Delete lines 100 through 180, and enter the source code below.

```

190 ; ARITHMETIC FLAG ACTIONS
200 START1      LD      A,33      ;A operand
210             LD      B,33      ;register operand
220             CP       B        ;33-33=0
230             LD      A,32      ;A operand
240             CP       B        ;32-33
250             LD      A,34      ;A operand
260             CP       B        ;34-33
270             END              ;end

```

This source code is similar to the subtract source code, except that it substitutes CP instructions for SUB instructions. Execute the program at low speed and watch how the Z flag and S flag change for the CPs. Also note that the A register doesn't change when the CP is executed.

The CP instruction is probably used more frequently than the SUB. Look upon it as virtually identical to the SUB — it uses the same addressing modes, register-to-register, immediate, register indirect with HL, and indexed.

The P(arity)/O(verflow) Flag

The parity/overflow flag, abbreviated P/V, is a dual-purpose flag. For arithmetic instructions like the ADD, SUB, and CP, it is used to record an **overflow** condition. Overflow occurs when the result of an add or subtract is too large to be held in 8 or 16 bits. Examples are an add of 240 and 20 in 8 bits, or a subtract of -30,000 from +10,000 in 16 bits. Both results will set the P/V flag to 1, indicating an overflow condition.

7 Adds, Subtracts, and Flags

For an example of the P/V flag operation, delete lines 190 through 270 and enter this code:

```
280 ; P/V FLAG OPERATION
290 ANUDR      LD          A,50          ;50
300           ADD          A,50          ;50+50=100=V!
310           LD          HL,-30000     ;-30000
320           ADD          HL,HL         ;-60000
330           LD          HL,30000      ;+30000
340           LD          BC,-2768      ;-2768
350           SCF              ;set carry
360           CCF              ;C=0
370           SBC          HL,BC         ;30000-(-2768)=32768
380           END                ;end
```

Set the speed to the lowest setting as before, assemble the program, and execute, keeping a sharp eye on the P/V flag.

What did you see? The first ADD added an immediate 50 to the 50 in the A register for a result of 100. This is not an overflow condition, and the P/V flag should have been set to a 0, indicating no overflow.

The next add added -30000 in the HL register to itself. This is an overflow condition and . . . If you were watching closely, you saw that the P/V flag **was not** set to indicate overflow for this instruction! Why not?

If you look at the flag settings in Appendix V, you'll see that the P/V flag and other flags are **not** affected by an ADD HL,XX. About the only flag affected is the C flag. Even though there was an overflow condition, the Z-80 does not provide an indication by setting the P/V flag. Another similar add, the ADC HL,XX, however, does change the P/V flag. Be aware of which instructions change the flags!

The SBC instruction subtracts the -2768 in the BC register pair from the 30000 in HL, yielding 32768, an overflow condition for 16 bits. In this case the P/V **is** set as the instruction is an SBC, a subtract with carry, and the flags **are** affected for the SBC.

Prior to the SBC, the Carry flag in the Flags register was set to a 1 by the SCF instruction (Set Carry Flag). It was then complemented by the CCF instruction (Complement Carry Flag), changing the 1 to a 0. The SCF and CCF are the only instructions that directly affect any flags. There is no "Reset Carry" instruction, so the SCF, followed by CCF, has the effect of a "reset carry." The Carry must be reset for the SBC to work properly, as the SBC "adds in" a possible carry from a previous subtraction. We'll discuss the Carry in this type of operation in greater detail in another lesson.

We didn't talk about the SBC in the previous lesson because of the interaction of the Carry Flag. If the Carry flag is reset (0), then the SBC simply subtracts the BC, DE, HL, or SP register pair from the contents of the HL register pair, with the result going to HL. The Z, S, and C flags are affected according to the results of the subtract. As in the ADD HL,XX, the operand for the subtract must be in another register pair and no other addressing modes are allowed.

Before we leave the discussion of the P/V flag, what about the second function, the "P" function? The P/V flag is used as a "Parity" flag for logical instructions and shifts and rotates, which we'll discuss in another lesson. For now, though we'll tell you that the parity function is simply a record of the number of 1 bits in an 8-bit result. If the number of 1 bits is even after certain instructions, then the P/V flag is set (1). If the number of 1 bits is odd after these instructions, then the P/V flag is reset(0). Check Appendix V to see which instructions affect the P/V flag for "parity" or "P."

The C(arry) Flag

The Carry flag is used in many different operations in the Z-80. Its original use was to hold the state of the carry from the most significant bit of an add or the borrow from the next bit on a subtract, as shown in Figure LESS7-2.

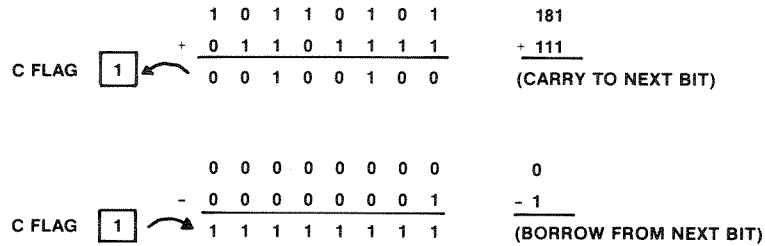


Figure LESS7-2. Carry Flag Use

Another use of the Carry flag is to hold the state (0 or 1) of the most significant bit on a “shift” or “rotate” operation. We’ll look at these applications of the Carry flag in future lessons.

To Sum It All Up

To review what we’ve learned in this Lesson:

- There are 6 flags in the Flags register; they are grouped together as the F register and form a register pair with the A register
- The Z flag is set after arithmetic and other instructions when the result is zero; it is reset when the result is non-zero
- The S flag is set after arithmetic and other instructions when the result is negative; it is reset when the result is positive
- The Z-80 acts as if two’s complement numbers were being processed in setting the S flag, but the arithmetic may be “absolute”
- The Compare instruction CP acts like a Subtract in Flag settings but does not put the result into the A register
- The P/V flag is set to indicate overflow conditions for both 8 and 16-bit operations
- The P/V flag is also used to record the “odd” or “evenness” of the number of 1 bits in a result for other instructions
- The Carry flag can be set by an SCF, or “complemented” by a CCF
- The Carry flag is used to record the carry or borrow from a high-order bit or the state of a bit on a shift or rotate
- The flags are set for some instructions but are not affected by others; the programmer must be aware of when they are affected

For Further Study

Appendix V: Flag Settings
SBC HL,XX instruction (Appendix V)



Lesson 8

Loops, Unconditional Jumps, and Conditional Jumps

Load LESS8 from disk.

Up to this point we've really only executed a series of sequential instructions. The power of a microprocessor or any computer, however, is being able to do "iterative" types of operations in "loops."

A loop is simply two or more cycles through the same set of instructions. For an example of a simple loop, enter the source code below, or use the existing source code from the Lesson file:

```
100 ; SIMPLE LOOP
110 SIMLOP      LD          BC,100      ;load BC with 100
120            LD          HL,0        ;clear HL
130 SIM010      ADD         HL,BC      ;add BC to HL
140            JP          SIM010      ;loop back
150            END
```

Assemble the code and check for errors. Now set the speed to a slow setting and execute by

ZX

As the program executes, you can see that the HL register pair is incremented by 100 each time through the loop. Notice also how the program counter display changes between the location of SIM010 and the instruction following. Hit BREAK to get back to the Command Mode.

The JP SIM010 instruction is an "unconditional" jump instruction. It **always** jumps back to a specified address.

Look at the three machine-language bytes for the JP SIM010 instruction. The first is an "opcode" byte of C3H. The next two are address bytes for the instruction.

The address bytes are in standard Z-80 address format, least significant byte followed by most significant byte. The value of the two address bytes correspond to the location of the label SIM010.

Symbolic Addressing

Using a label instead of an absolute address in memory is termed "symbolic addressing." It relieves the programmer of having to compute the actual address for the Jump. Of course, you could still "hand assemble" machine-language object code, but it's much more convenient to let the assembler do it for you.

We've used labels for other source code in previous lessons, but this is the first time that we're using them for their primary purpose — "tagging" an instruction for a Jump point.

This assembler and all assemblers build a table of symbols (appropriately enough called a "symbol table"). In it are all labels and other symbols encountered in the program. The assembler uses the symbols to "build" the addresses in instructions, as, for example, the address of SIM010 in the program above.

You can use labels for locations anytime you wish. The labels are usually associated with Jump points but do not have to be. Labels may be 1 to 6 characters, the first of which starts with an alphabetic character. One convention that I've used here, and that other programmers often use, is to make the primary label a 6-character descriptive label and to make following labels the first 3 characters of the "module," followed by 3 digits. Often the digits will be in order, as in BASIC lines. You might have SQROOT as the first label of a square root code segment, for example, and the labels following would be SQR010, SQR020, SQR080, and SQR090.

8 Loops, Unconditional Jumps, and Conditional Jumps

Unconditional Jumps

The JP instruction above is one of four “JP” unconditional jumps in the Z-80 (there are other “JR” jumps that we’ll talk about in the next lesson). It always has the same format of a C3H opcode, followed by two address bytes, and always transfers control to the jump address. Note that the jump address can be anywhere in memory — location 0 through 65,535; of course some of these addresses are invalid in the TRS-80s.

There are three other “JP” unconditional jump instructions, and they all use a 16-bit register as a “register indirect” pointer. They are

JP (HL)	;jump indirect to HL
JP (IX)	;jump indirect to IX
JP (IY)	;jump indirect to IY

You can guess how these work. The HL, IX, or IY holds the Jump address, and the JP simply looks at the address from the register and jumps to that location. Here’s an example: Delete lines 100 through 150 and enter this program:

```
160 ; INDIRECT JUMPS
170 SYMBL1    LD      HL,SYMBL2    ;load location
180          JP      (HL)          ;jump
190          LD      A,1           ;dummy
200 SYMBL2    LD      IX,SYMBL3    ;load location
210          JP      (IX)          ;jump
220          LD      A,1           ;dummy
230 SYMBL3    LD      IY,SYMBL4    ;load location
240          JP      (IY)          ;jump
250 SYMBL4    LD      A,1           ;dummy
260          END                  ;end
```

Set the speed to slow and execute the program. You should see the LD A,l instructions bypassed as the program jumps first to SYMBL2, then to SYMBL3, and then to SYMBL4.

By the way, when the assembler assembled this source code, how did it know what value to use in the immediate instructions for loading HL, IX, and IY? The answer is that it went to the symbol table and tried to find a “match” for the three symbols. As all three symbols were labels, they were in the symbol table, and the assembler found corresponding address values for each of the three. It then filled in these address values in the immediate instructions.

The values we used for immediate instructions up to this point have been numeric values in decimal or hexadecimal, but they can be “symbolic” as well, as you can see. As a matter of fact, symbolic references such as these are quite common in assembly-language programs, and you will be using them all the time as you do more of this type of programming.

Could you have used numeric values? Sure, but you wouldn’t know what the values were until after you assembled. You’d have had to make up some dummy values, do a single assembly pass, find out the actual locations, and then fill in the proper addresses. With symbolic addressing, the assembler does it for you.

Conditional Jumps

The loop in the program above has one drawback. It never stops. We can easily control the loop, however, with a “conditional” jump. As an example of a loop with a conditional jump, delete lines 160 through 260 and enter this code:

```

270 ; LOOP WITH CONDITIONAL JP
280 ADDNUM    LD      HL,0           ;zero HL
290          LD      A,100          ;counter
300          LD      B,0           ;0 to B
310 ADD010    LD      C,A           ;A now in BC
320          ADD     HL,BC          ;add 100+99+98...
330          SUB     1              ;A-1 to A
340          JP      NZ,ADD010      ;loop if A not 0
350          END

```

Assemble this code without any errors and execute the program. If you execute at moderate speed, you will see the loop from ADD010 through the JP NZ,ADD010 repeat 100 times. The first time, 100 is added to HL (initially set to 0), the next time, 99 is added, the next 98, and so forth, down to 0.

Let's look at the code in detail. The instruction at line 280 loads the HL register pair with 0. HL will be used to hold the running total.

The next instruction loads the A register with 100. A will be used to hold the current number and will start with 100 and "decrement" down to 0.

The next instruction zeroes the B Register.

The loop starts at ADD010. Each time through the loop, the following actions occur:

- The C register is loaded with the contents of A. Since the B register always contains 0, the BC register, taken as a whole, contains the value of A.
- The BC register is added to the contents of HL with the result going into HL.
- A SUB A,1 is done to subtract 1 from the count in A. A is initially 100. After the first SUB, it is 99, after the second, it is 98, and so forth.
- The Z flag in the Flags is set on the result of the SUB 1. If the Z flag is reset (NZ) then A is not 0; if the Z flag is set (Z), then A is 0.
- The JP NZ instruction tests the Z flag. If it is not set (NZ), the jump is made just as if the JP was an unconditional JP. If the Z flag is set (Z), then the JP "falls through" to the end instruction and 100 passes through the loop have been made.

Conditions for JP

We used an "NZ" for the condition in the JP. This is equivalent to "Jump if Zero Flag Reset." Logically, this is the same as "Jump if Non-Zero." The mnemonics for a conditional JP may be somewhat confusing, so we'll list them here:

Mnemonic	Meaning	Flag Setting for JP
Z	Jump if Zero	Z=1
NZ	Jump if non-Zero	Z=0
C	Jump if Carry	C=1
NC	Jump if no Carry	C=0
M	Jump if Minus	S=1
P	Jump if Positive	S=0

Although it might be more convenient to use a JP SZ,XXXX, for "Jump if Sign Zero," you'll have to remember all of the above mnemonics — they're the ones that this assembler, and most Z-80 assemblers, recognize.

8 Loops, Unconditional Jumps, and Conditional Jumps

Here are some examples of the use of these mnemonics in JP instructions:

SUB	B	;A-B
JP	Z,ZERO	;jump if A=B
JP	P,PLUS	;jump if A>B but not 0
JP	M,MINUS	;jump if A<B

In the first of these instructions, B is subtracted from A.

The result is either 0, greater than zero, or negative. The JP Z tests the zero flag. If the Z flag is set (Z condition), then a JP is made to location ZERO.

If the JP P instruction is executed, then A cannot equal B. A test is made of the S(ign) flag by the mnemonic “P” for positive. If the S flag is reset (P condition), then a JP is made to location PLUS.

If the JP M instruction is executed, then A cannot equal B or be greater than B. As a matter of fact, the result must be negative here, and the S flag must be set (M). The JP M, MINUS always results in a JP!

There is also a conditional JP on the Parity/Overflow flag. We’ll discuss this one in another lesson. Also, there are a number of “JR” conditional jumps that we’ll cover in the next lesson.

A Comparison Test Using Modify Memory

Let’s expand the code above into a full-fledged comparison test of two memory locations. Delete lines 270 through 350 and enter the following source lines, or use the Lesson File.

```
360 ; COMPARISON TEST OF B000H AND B001H
370 LD A,(B001H) ;get second operand
380 LD B,A ;now in B
390 LD A,(B000H) ;get first operand
400 SUB B ;A-B or (B000H)-(B001H)
410 JP NZ,NEXT1 ;go if not 0
420 JP STORE ;wind up
430 NEXT1 JP M,NEXT2 ;go if A<B
440 LD A,1 ;1 to A
450 JP STORE ;go to store
460 NEXT2 LD A,OFFH ;-1 to A
470 STORE LD (B002H),A ;store
480 END ;end
```

Assemble the program and get an error-free assembly.

The program compares two operands in the “experiment area” at locations B000H (operand 1) and B001H (operand 2). The result of the comparison is put into location B002H. The result will be as follows:

Condition	Result (B002H)
(Op 1=Op 2)	0
(Op 1>Op 2)	1
(Op 1<Op 2)	-1

To run the program you must first put two operands you want to compare into locations B000H and B001H. Do this by using the ZT and ZM commands as follows:

```
ZT B000 (ENTER)
ZM B000=xx nn
B001 xx mm (ENTER)
```

The ZT B000 “traces” or displays the B000H area.

Entering ZM B000 will display the contents of that location. After the display, enter the hexadecimal value (nn) you want put into location B000H, followed by a space. ALT will then display “B001” followed by the contents of this location. Enter the hexadecimal value (mm) you want put into location B001H, followed by an ENTER.

After you have entered the values check the trace display to see that the data has been changed properly.

Now execute the program by

ZX

At the end of execution, you should see the contents of location B002H changed to reflect the results of the comparison. It should be a 0, 1, or -1 (equals, greater than, or less than).

The program works similarly to the earlier version, and we’ll leave it up to you to scrutinize it.

To Sum It All Up

To review what we’ve learned in this lesson:

- Loops are portions of code that are executed more than once in sequence
- A label used in a source line generally gives the line a “name” that the assembler will reference for jump addresses
- Labels are 1 to 6 characters long and start with an alphabetic character
- There are four “unconditional” jumps in the Z-80 that always jump to the jump address — JP (address), JP (HL), JP (IX), and JP (IY)
- Labels can also be referenced for loading immediate data where the data is an address, such as a jump address
- “Conditional” jumps jump if the condition is met, but otherwise do nothing
- Conditional jumps test the state of the Zero flag, the Sign Flag, the Carry Flag, or the P/V Flag
- Conditional jumps use the following mnemonics: Z, NZ, C, NC, M, P

For Further Study

Appendix V — check the flag actions for conditional and unconditional jumps

Use the ZM command to modify other memory locations in the experiment area (B000H) with various values.



Lesson 9

Relative Jumps, Conditional and Unconditional

Load LESS9 from disk.

In the last Lesson we looked at unconditional and conditional “JP” type jumps. This lesson we’ll investigate another type of jump, called the “relative” jump. The mnemonic for this jump is “JR,” for “Jump Relative.”

To see the basic difference between the two jumps, enter the source code shown below, or use the existing source code from the Lesson File. (This example does not have a “relative jump” yet, but it will eventually, so don’t be confused by the “JP”)

100 ; RELATIVE JUMPS			
110 RELJRS	LD	HL,(0B000H)	;load square
120	LD	A,-1	;clear square root
130	LD	BC,-1	;initialize odd integer
140 RELO10	ADD	A,1	;square root+1
150	ADD	HL,BC	;square-odd integer
160	DEC	BC	;BC-2
170	DEC	BC	;square-odd integer
180	JP	C,RELO10	;loop if not minus
190	LD	(0B002H),A	;store square
200	END		;end

Assemble the source code until you get an error-free assembly.

The program above ties together a lot of the concepts that we have discussed in previous lessons into a program that will calculate square roots. As you know by now, the Z-80 doesn’t have the capability to even multiply and divide numbers; developing a square root program is therefore not a minor accomplishment.

Let’s see how the program works: A square root of a number is a number which when multiplied by itself will give the number, in case you’re rusty. The square root of 100, for example, is 10, as 10 times 10 is 100. The square root of 169 is 13, as 13 times 13 is 169. The square root of 178 is 13.34.

One way to find a square root is to take the “square” and start subtracting “odd integers” — 1, 3, 5, 7, 9, and so forth from it. The number of subtracts that can be made is the square root. Don’t ask me how it works, but it does!

Take a square of 102, for example. 102-1 is 101-3 is 98-5 is 93-7 is 86-9 is 77-11 is 66-13 is 53-15 is 38-17 is 21-19 is 2-21 is -19. We were able to subtract 1, 3, 5, 7, 9, 11, 13, 15, 17, and 19 from 102, a total of 10 odd integers, so the square root is 10. In this method we don’t get the fractional part of the square root, only the “integer part.”

To run the program, use the ZT command to trace the B000H area and the “Modify Memory” ZM command of ALT to enter a square into locations B000H and B001H. Don’t forget that the two bytes of the number must be in Z-80 address format, least significant byte followed by most significant byte. The number 1000, for example, would be E8H, followed by 03H.

After entering the value and verifying that it is correct, run the program by

ZX

At the end of the program, you’ll see the integer square root in location B002H. If you used a 1000 as the square, for example, you’ll see 31 decimal or 1FH in location B002H.

9 Relative Jumps, Conditional and Unconditional

Let's review the steps of the program:

First, HL was loaded with the square from location B000H and B001H. This was a "direct load" of two bytes.

Next, the A register was loaded with -1. The A register will have 1 added to it each time through the loop of the program. We must start off with -1 so that the first add of 1 results in 0.

Next, the BC register pair was loaded with -1. The BC register pair holds the "odd integer" of 1, 3, 5, and so forth. BC will have 2 subtracted from it each time through the loop. The odd integer in BC will be added to the square in HL each time through the loop. An add of a negative value is the same as a subtract of a positive number, after all.

The loop starts at REL010. You can see the indented comments that indicate the instructions that are part of the loop.

Each time through the loop, these things happen:

One is added to the A register. This "bumps" the count of the number of odd integers successfully subtracted from the square.

An ADD HL,BC is done. This "subtracts" an odd integer in BC from the square (or from the "residue" of the last subtract) in HL.

Two is subtracted from the odd integer in BC by two "DEC BC" instructions to get the next odd integer. (We'll discuss DEC in the next lesson.) We started off with +1, and after the first subtract we have -1.

The Carry flag is set to 1 (C) if the result of this subtract is 0 or greater. In other words, as long as the result in HL is not a negative number, the C flag will be set. As soon as the result "goes negative," the C flag will be reset (NC or 0). Why is the carry set in this fashion? There's no easy answer to this. The Carry flag is simply set from the carry bit out of HL as the add is done, and one of the peculiarities of an add is that there will be a carry as long as the result is not negative. You don't have to memorize this fact, but bear in mind that the Carry flag can be used to test for this condition.

If the result has not "gone negative," the JP is made back to REL010 for the next operation. If the result has gone negative, A contains the count of odd integers successfully subtracted, and this is put into location B002H.

Try running this program at slow speed and observing the registers. You'll be able to see what is happening in the loop quite easily. Also, you might want to refer to Figure LESS9-1, a "flowchart" of the program. It represents the program "flow" in schematic form.

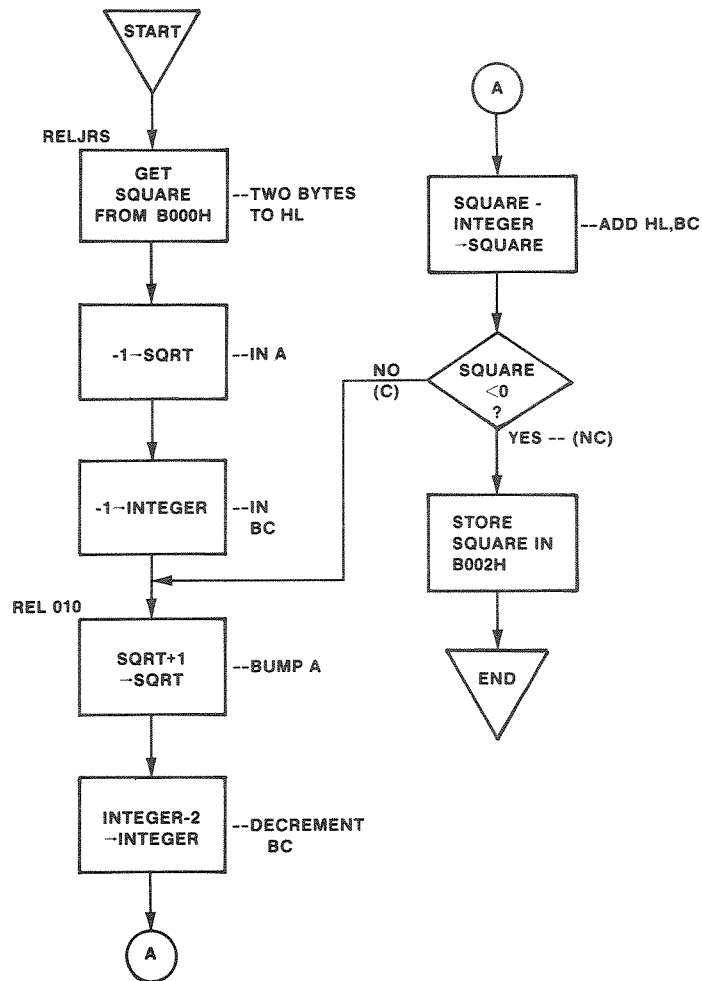


Figure LESS9-1. Square Root Flowchart

Using a JR in Place of a JP

The JP C,REL010 instruction used one of the conditional JPs we discussed in the last chapter. Look at the JP and note the machine-language code for it. It consists of 3 bytes, an “opcode,” followed by two address bytes that are the address of label REL010.

We’re now going to replace the JP with a JR. We can do this easily with the Editor by the following sequence:

```

D180          (deletes line 180)
I180,1        (starts insert mode)
00180        JR    C,RELO10  (ENTER)
(BREAK)
  
```

After you’ve gone through the above sequence, you should have the same source code as before, except that the “JP” is a “JR.” You should have:

9 Relative Jumps, Conditional and Unconditional

100 ; RELATIVE JUMPS			
110 RELJRS	LD	HL,(0B000H)	;load square
120	LD	A,-1	;clear square root
130	LD	BC,-1	;initialize odd integer
140 RELO10	ADD	A,1	;square root+1
150	ADD	HL,BC	;square-odd integer
160	DEC	BC	;BC-1
170	DEC	BC	;BC-2
180	JR	C,RELO10	;***REPLACED***
190	LD	(0B002H),A	;store square
200	END		;end

When you have made the changes, assemble the source code until you get an error-free assembly. After assembly, look at the machine code for the JR. It should be 38 F9 in hexadecimal. Notice anything different about the JR codes compared to the JP?

The JR instruction is 2 bytes long, while the JP is 3 bytes! And that's the reason the JR was added to the Z-80 instruction set — primarily to save memory. The JP was used in the 8080 microprocessor; the Z-80 kept the original 8080 instruction set and added some niceties, and the JR instructions were one of the improvements. As jumps are used all the time, saving one byte in a jump can result in a 5% or more savings in memory space for a program.

Relative Addressing

Look again at the JR bytes. The first byte is an “opcode” byte that tells the Z-80 that a JR is about to be executed. The next byte somehow must specify the address for the jump. But how?

The JR uses “relative addressing.” Relative to what? Relative to the location of the instruction. Here's the way the Z-80 finds the address for the jump (see Figure LESS9-2):

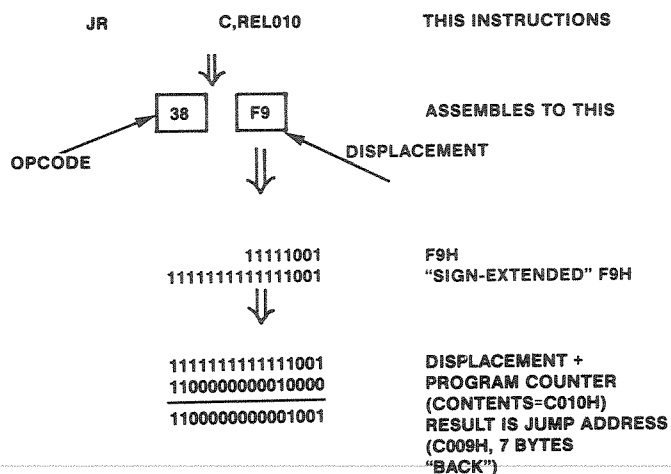


Figure LESS9-2. Relative Addressing

The second byte of the JR instruction is made into a 16-bit number. This byte is an 8-bit “signed value” (negative or positive) with the first bit representing the sign. Our old friend (enemy?) the two's complement representation is in force here.

To make an 8-bit two's complement number into 16 bits, we have to “sign extend” the sign bit. If the sign bit is a 0, we put zeroes into the most significant byte. If the sign bit is a 1, we put ones into the most significant byte. In this case we'd need ones.

This 16-bit number is then added to the contents of the Program Counter register. The Program Counter always points to the next instruction to be executed. In the case of the JR, it points to the LD (0B002H),A instruction.

Adding the PC and the 16-bit number together gives the location of the jump address, as shown in the figure. And all that with just 8 bits for an address!

Of course, all of this is done internally in the Z-80. You never have to do the arithmetic as we just did. The assembler will take care of putting in the proper value in the JR instruction, so that all you have to do is use a symbolic label for the JR, as we did.

Types of JRs

There are four conditional JRs and one unconditional JR. The four conditional JRs are

• JR C,XXXX	Jump if Carry
• JR NC,XXXX	Jump if no Carry
• JR Z,XXXX	Jump if Zero
• JR NZ,XXXX	Jump if not Zero

The JRs work exactly the same as the JPs we discussed in the last chapter. Note that there are no JRs for the Sign Flag or Parity/Overflow Flag as there were JPs. The four JRs above, though, are the most commonly used conditional jumps.

Limitations of JRs

There is one slight hitch, however, in using JRs. As there is only one byte for the address, the jump “range” is limited. You know that in an 8-bit two’s complement number we can hold values of -128 through +127. As the second byte is the “relative” jump address in two’s complement form, we can, therefore, jump only 128 locations back and 127 locations forward.

And don’t forget that those numbers are referenced to the Program Counter, which points to the instruction after the JR. Referenced to the JR, then, we can only jump back 126 bytes or forward 129 bytes.

That’s no problem, however. If we try to use a JR and jump “out of the range,” the Assembler will detect the error and give us an error message. We can then change the JR to a JP and everything will be fine.

Use the JR in place of the JP for an unconditional jump, or for a Carry or Zero conditional jump, whenever you’re jumping a short ways away.

A Special Relative Jump

Now we’ll look at a special form of a “relative” jump. Delete lines 100 through 200, and enter the following program, or use the source code from the Lesson File:

```

210 ; DJNZ USE
220 ADDNUM    LD      HL,OB000H    ;point to B000H
230          LD      A,0          ;clear total
240          LD      B,10         ;initialize counter
250 ADDO10    ADD     A,(HL)       ;add in number
260          INC     HL           ;point to next
270          DEC     B            ;B-1 to B
280          JR      NZ,ADD010    ;loop if not zero
290          END                ;end

```

This program adds the contents of memory locations B000H through B009H together in the A register.

9 Relative Jumps, Conditional and Unconditional

(The maximum sum will be $255+255+255+\dots+255$ or 2550 — too large to hold in 8 bits. We'll assume, though, that all numbers are in the range of 0 through 255, which will allow us to hold the result in A.)

Here's how the program works:

The HL register pair is loaded with a value of B000H. HL is used to point to the B000H area.

The A register will hold the total, and it is cleared.

The B register holds the count of the number of times through the loop. It is initially set to 10 and will be "counted down" to 0.

The loop starts at ADD010. Each time through, the contents of the next memory location from the B000H area is added to the A register contents and the HL register pair pointer is incremented by 1 to point to the next location.

The next instruction decrements the B register. The Z flag is set to Z if the decrement of B resulted in zero, or is reset (NZ) if the decrement of B resulted in non-zero.

The next conditional JR jumps back to the beginning of the loop if the result is non-zero (NZ).

A total of 10 passes is made through the loop to add the contents of the 10 locations. The count in B determines the end of the loop when it reaches 0.

Assemble and execute the program after putting some small values in locations B000H through B009H. The result will be in A at the end of the program.

Now delete lines 210 through 290, and enter this code:

```
300 ; DJNZ USE
310 ADDNU1    LD      HL,0B000H    ;point to B000H
320          LD      A,0           ;clear total
330          LD      B,10          ;initialize counter
340 ADD011    ADD     A,(HL)        ;add in number
350          INC     HL            ;point to next
360          DJNZ    ADD011        ;loop if not zero
370          END                  ;end
```

Assemble the above program. Look at the DJNZ instruction machine code bytes. Note that it is a "relative" type of instruction, just like the JRs.

The DJNZ replaces the DEC B and JR NZ,ADD010 instructions of the previous example with one instruction. It does exactly the same thing as the previous two instructions, decrementing the contents of the B register by 1, and then testing the Z flag to jump if non-zero (NZ).

The DJNZ can be used anytime that a "counter" is being decremented down to zero in the B register for loop control. Many loops use less than 255 "iterations," or passes through the loop, and the DJNZ is perfect in these cases. The loop will repeat for the count in B; if B is 0 initially, 256 passes will be made, as the jump test is made **after** the decrement.

To Sum It All Up

- Relative jumps are 2 bytes instead of 3 bytes as in JPs, saving memory space
- Relative jumps use a two's complement one-byte relative address which will give the jump address when added to the Program Counter
- There is one unconditional JR and four conditional JRs that work with C, NC, Z, and NZ

- The range of a JR is limited to 126 locations back or 129 locations forward from the JR; this is usually enough
- A DJNZ combines a decrement of the B register and a JR NZ instruction for loop control (up to 256 passes)

For Further Study

Use JRs in modifications of the above programs and examine their address bytes — private study



Lesson 10

Other Arithmetic and Logical Operations

Load LESS10 from disk.

Up to this point we've covered many of the common arithmetic operations that you can do in the Z-80. In this lesson we'll try to "fill in the gaps" and cover increment and decrement operations and logical operations.

Increments and Decrements

We've already used the DEC and INC instructions in several examples. "DEC" stands for "Decrement," and INC stands for "Increment." An increment adds 1 to the contents of a register or memory location, while a decrement subtracts 1 from the contents of a register or memory location.

Increments and decrements exist because adding 1 or subtracting 1 is a very common operation in assembly-language code. In the case of using the HL register pair for accessing sequential data, for example, it's much easier to do an "INC HL" than to load BC with 1 and add the BC register to HL as in

LD	BC,1	;prepare to increment
ADD	HL,BC	;increment

There are really two sets of INCs and DECs. The first set works on an 8-bit register or memory location and uses the same addressing modes as an ADD or SUB. Here are some possibilities:

INC	A	;increment A
INC	C	;increment C
DEC	E	;decrement E
INC	(HL)	;increment memory
DEC	(IX)	;decrement memory
INC	(IY)	;increment memory

The first three instructions increment the A register, increment the C register, and decrement the E register, respectively. The next instruction increments the memory location pointed to by HL. The last two instructions are similar, but use either IX or IY as a pointer to a location in memory.

Notice that the addressing modes are identical to the ADD or SUB for 8-bit registers, except that there is no "immediate" mode, for example, "INC A,1." The reason for this, of course, is that the instruction is predefined to increment by 1 and needs no other data.

Eight-bit increments and decrements affect the Flags about the same way that ADDs and SUBs do. A zero result after a decrement sets the Z flag, for example. We said "about the same way" because the Carry flag is not affected after an increment or decrement. It remains the same.

The second set of INCs and DECs operate on 16-bit register pairs. The following instructions are included in this set:

DEC	BC	INC	BC
DEC	DE	INC	DE
DEC	HL	INC	HL
DEC	SP	INC	SP
DEC	IX	INC	IX
DEC	IY	INC	IY

The increments and decrements in this set increase or decrease the given register by 1 count. The result, of course, goes back into the register. **None of the INCs or DECs affect any Flags!** This last point is very important because it means that you can't increment or decrement a 16-bit register pair and check for a 0 result!

10 Other Arithmetic and Logical Operations

Increments and decrements of either the 8- or 16-bit type are really “unsigned” operations. The value after the increment or decrement can’t be thought of in two’s complement terms.

For example, suppose that we’re incrementing the B register. We’ve started at 0 and have gone up — 1, 2, 3 ... until we’re now at 127 decimal or 01111111 binary. What is the next increment? Adding one produces the value 10000000, which in absolute form is 128 but in two’s complement form is -128! Always think in terms of 0 through 255 on increments and decrements and you’ll be fine.

When the maximum count for 8 or 16 bits is reached, the next increment “resets” the count to 0. 11111111, for example, becomes 00000000. Decrementing from 0 produces the maximum count, which then counts down to 0 again. 00000000 becomes 11111111 and then 11111110, and so forth.

Enter the following source code or use the existing code from the Lesson File. This code will let you see increment and decrement action on 8 bits and will also let you observe the Flag changes. Execute it after an error-free assembly.

```
100 ; 8-BIT INCREMENTS AND DECREMENTS
110 START      LD      A,0           ;start at 0
120 STAO10      INC      A           ;bump A
130            JR      NZ,STAO10     ;loop til 0
140 STAO20      DEC      A           ;decrement A
150            JR      NZ,STAO20     ;loop til 0
160            END                ;end
```

The program is very simple. It increments A from 00000000 through 11111111 and back to 00000000 again, and then decrements 00000000 to 11111111 and then counts down to 00000000 again. Run it at a faster speed (unless you like 512 iterations) and observe the Flag changes. The S flag will echo the sign bit in bit 7, while the Z flag will be set only on A=0 (2 times in the program).

The NEG and CPL Instructions

The NEGate and ComPLement instructions are two instructions that operate only on data in the A register. No other addressing modes are permitted.

The CPL instruction takes the contents of the A register and “complements” it by changing all ones to zeroes and all zeroes to ones.

Suppose that the A register contained

00111111

After a CPL instruction had been done, the new contents would be:

11000000

None of the “conditional” flags we’ve talked about are affected by a CPL, although the N and H Flags are set to a 1.

The NEG instruction takes the contents of the A register and “two’s complements” it, changing all ones to zeroes and all zeroes to ones and adding one. We’ve already seen how the two’s complement works in an earlier lesson. The NEG simply performs the two’s complement conversion automatically. The effect of this is to “negate” a number, changing a positive number into a negative number and vice versa.

Suppose that the A register contained:

00111111, or decimal 63

After a negate, the A register would contain:

11000001,

or decimal -63 (in two's complement).

The NEG sets the Flags just as in a SUB instruction. If the result was 10000000, for example, the Z flag would be reset (NZ) and the S flag would be set (M).

The CPL and NEG are used infrequently compared to adds, subtracts, decrements, and increments, but they come in handy from time to time.

Logical Operations

The Z-80 has three instructions that perform logical operations, the OR, AND, and exclusive OR (XOR).

If you've done some BASIC programming, you'll be familiar with the first two logical operations, and the XOR is simply a variation.

All of the instructions use the same registers and addressing modes as a SUB. They all use one operand from the A register and a second operand from another cpu register or from memory. The result goes back into the A register, and the Flags are affected.

An OR takes the 8-bit value in A and ORs it with the second operand. An OR operates at a "bit-level" — one bit at a time with no bit affecting any other bit. For each bit:

```
0 OR 0=0
0 OR 1=1
1 OR 0=1
1 OR 1=1
```

A bit in the result is set, then, when either one OR the other bit OR both are set. The result of a typical OR might be:

```
  00111010
OR 01010111
-----
  01111111
```

Only in the case of the most significant bit was the result bit not 0, and that was because both operand bits were 0.

ORs are typically used to set a single bit in the middle of other bits. Suppose you wanted to set bit 5 of the C register. (Bit positions are 7, 6, 5, 4, 3, 2, 1, and 0 from left to right.) One way to do it would be:

```
LD      A,C      ;get C register
OR      20H      ;set bit 5
LD      C,A      ;restore C
```

One important use of OR: OR A SETS THE FLAGS DEPENDENT UPON WHAT IS IN A AND CLEARS THE CARRY FLAG. IT CAN BE USED EITHER TO TEST A WITHOUT A COMPARE OR TO RESET THE CARRY FLAG OR TO PERFORM BOTH ACTIONS*

An AND takes the 8-bit value in A and ANDs it with the second operand. An AND also operates at a "bit-level" — one bit at a time with no bit affecting any other bit. For each bit:

```
0 AND 0=0
0 AND 1=0
1 AND 0=0
1 AND 1=1
```

A bit in the result is set, then, when one bit AND the other bit are set. The result of a typical AND might be:

10 Other Arithmetic and Logical Operations

```
  00111010
AND 01010111
-----
  00010010
```

The only result bits that were set were ones in which both operand bits were ones.

ANDs are typically used to “mask” data. Suppose you wanted to find the setting of bits 1 and 0 of the C register. One method would be:

```
LD      A,C      ;get C register
AND     3         ;get bits 1 and 0
```

At the end of these two instructions, A would contain 000000XX, where XX are the settings of bits 1 and 0 in the C register. The bits in bit positions 1 and 0 “fell” through on the AND, and the other bits were “masked out”:

```
  01010010 (C REGISTER)
AND 00000011 (AND VALUE)
-----
  00000010 (RESULT OF AND)
```

An XOR takes the 8-bit value in A and XORs it with the second operand. An XOR again operates at a “bit-level” — one bit at a time with no bit affecting any other bit. For each bit:

```
0 XOR 0=0
0 XOR 1=1
1 XOR 0=1
1 XOR 1=0
```

A bit in the result is set, then, when either one OR the other bit but NOT both bits is set. The result of a typical XOR might be:

```
  00111010
XOR 01010111
-----
  01101101
```

Whenever the operand bits are both 0s or both 1s, the result bit is a zero.

XORS are used infrequently compared to ORs and ANDs. One classic example of the use of an XOR is to check the sign of a result. Suppose that we were going to multiply two 8-bit numbers. The sign of the result could be determined by:

```
  10101010 (-86 DECIMAL)
XOR 01010100 (+84 DECIMAL)
-----
  11111110 (XOR RESULT)
```

The result sign of the XOR is a 1, so the sign of the multiply result will be a 1, or negative.

A special case of XOR is XOR A. What does this instruction do?

XOR A clears the A register to 0 and sets the Z Flag to 0. It should be used as an “efficient” clear 0, as it is only one byte.

A Sample Problem Using ANDs and ORs (or ORs or ANDs)

Suppose that we have 8 bytes in the 8 memory locations of B000H through B007H. Each byte represents data on one employee of the ACME Z-80 Programming School.

The data has been compressed down into a few codes and “fields” within the byte, as shown in Figure LESS10-1.

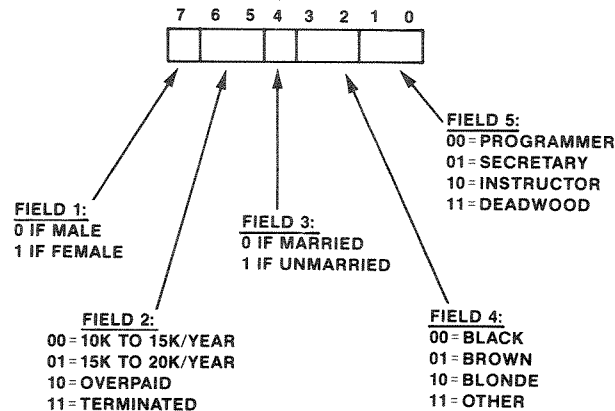


Figure LESS10-1. Fields

Bit 7 of each employee byte is a 0 if male, 1 if female.

Bits 6 and 5 are the pay range:

- 00 is between \$10,000 per year and \$15,000 per year
- 01 is between \$15,000 per year and \$20,000 per year
- 10 is marked "overpaid"
- 11 is terminated

Bit 4 is a 0 if married, 1 if unmarried.

Bits 3 and 2 are hair color:

- 00 is black
- 01 is brown
- 10 is blonde
- 11 is other

Bits 1 and 0 are the occupation code:

- 00 is programmer
- 01 is secretary
- 10 is instructor
- 11 is deadwood

Here's the problem: Can you write a program to find the number of employees that are overpaid deadwood and set the terminate flag?

Use the techniques you've learned so far in indexing the data via HL indirect addressing. Use an AND to get the proper fields. If you've found an employee that is both overpaid and deadwood, set bits 6 and 5 to "terminate" him or her. Before you start writing, delete lines 100 through 160 and enter the following program:

10 Other Arithmetic and Logical Operations

```
170 ; SAMPLE DATA FOR ACME PROGRAMMING SCHOOL
180 STDATA      LD      HL,0B000H      ;point to data buffer
190            LD      DE,DATA        ;point to program data
200            LD      B,8             ;8 employees
210 STD010      LD      A,(DE)         ;get byte
220            LD      (HL),A         ;store
230            INC     HL              ;increment buf pointer
240            INC     DE              ;increment data pntr
250            DJNZ    STD010          ;loop if not 8
260            JR      ENDLOC          ;go to end
270 DATA      DEFB     43H            ;data
280            DEFB     17H
290            DEFB     0AH
300            DEFB     56H
310            DEFB     0C3H
320            DEFB     30H
330            DEFB     00H
340            DEFB     01H
350 ENDLOC      END                    ;end
```

Assemble and execute the program. It will initialize the data in the B000H area. Now see if you can write a program to terminate that deadwood!

When you have a program, delete lines 170 through 350 and enter your program. Edit and assemble it until you get an error-free assembly. Be certain to include an END statement at the end.

Hint: Look at previous programs for instructions. You should be able to find everything you need there. At the end of the program, you should set the terminate codes in all applicable employee bytes. Execute your program. If you have problems refer to the operating procedures for ALT in the first four lessons and Appendix V. Sorry, we can't help you any more than that. You're on your own here!

Are you done? If it took a long time, don't despair. The first program is terrible! They'll get much easier as you go along.

The before and after area is shown in Figure LESS10-2. If you didn't come up with the proper results, you may have many disgruntled employees of ACME. Here's one of many versions that will work:

```

100 ; TERMINATE PROGRAM
110 TERMPR      LD      HL,OB000H      ;point to employee area
120             LD      B,8             ;8 employees
130 TERO10      LD      A,(HL)          ;get employee data
140             AND      3              ;get occupation
150             CP       3              ;is it "deadwood"?
160             JR       NZ,TERO20      ;go if not
170             LD      A,(HL)          ;get data again
180             AND      60H           ;get salary
190             CP       40H           ;is it "overpaid"?
200             JR       NZ,TERO20      ;go if not
210             LD      A,(HL)          ;get data byte
220             OR       60H           ;set bits 6,5 to 11
230             LD      (HL),A          ;store modified byte
240 TERO20      INC      HL             ;point to next
250             DJNZ     TERO10         ;go if not 8
260             END
;end

```

43H	7000H	0	1	0	0	0	0	1	1	MALE, OVERPAID, MAR, BLK HR, DEADWOOD
17H	1	0	0	0	1	0	1	1	1	MALE, 10K-15K, UNM, BRWN HR, DEADWOOD
0AH	2	0	0	0	0	1	0	1	0	MALE, 10K-15K, MAR, BLONDE, INSTR
56H	3	0	1	0	1	0	1	1	0	MALE, OVERPD, UNM, BROWN HR, INSTR
C3H	4	1	1	0	0	0	0	1	1	FEMALE, OVERPD, MAR, BLK HR, DEADWOOD
30H	5	0	0	1	1	0	0	0	0	MALE, 15K-20K, UNM, BLK HR, PROG
00H	6	0	0	0	0	0	0	0	0	MALE, 10K-15K, MAR, BLK HR, PROG
01H	7	0	0	0	0	0	0	0	1	MALE, 10K-15K, MAR, BLK HR, SECRE

63H	1	0	1	1	0	0	0	1	1	TERMINATED!
17H	2	0	0	0	1	0	1	1	1	
0AH	3	0	0	0	0	1	0	1	0	
56H	4	0	1	0	1	0	1	1	0	
E3H	5	1	1	1	0	0	0	1	1	TERMINATED!
30H	6	0	0	1	1	0	0	0	0	
00H	7	0	0	0	0	0	0	0	0	
01H	8	0	0	0	0	0	0	0	1	

Figure LESS10-2. Before and After Results

To Sum It All Up

To review what we've learned in this lesson:

- Increments and decrements add or subtract 1 from an 8-bit register or memory location, or from a 16-bit register
- Flags are set for 8-bit increments and decrements, but not for 16-bit increments and decrements
- The CPL "complements" A register data, changing all 0s to 1s and all 1s to 0s
- The NEG takes the two's complement of A register data

10 Other Arithmetic and Logical Operations

- ANDs, ORs, and XORs work with one operand in A and one from another CPU register or memory
- ANDs set the result bit to a 1 only if both operand bits are 1
- ORs set the result bit to a 1 if either operand bit is a 1
- XORs set the result bit to a 1 if either but not both operand bits is a 1

Lesson 11

Subtracts with Carry and Multiple-Precision

Load LESS11 from disk.

In this Lesson we're going to look at ADC and SBC, adds and subtracts with "Carry." These instructions are very similar to the standard ADDs and SUBs, except that the state of the Carry flag is added in or subtracted. ADC and SBC allow "multiple-precision" operations which can extend the range of processing to any size number, not just 8 and 16 bits. To understand the ADC and SBC, we first have to look at "multiple-precision" numbers.

Multiple-Precision Numbers

A multiple-precision number is a fancy term for any integer number format that is larger than the size the microprocessor can handle with its built-in instructions.

In the Z-80, we can add 8 and 16-bit operands. The maximum number that can be represented in 8 bits is 255 (unsigned), while the maximum number that can be represented in 16 bits is 65,535 (unsigned). What about large numbers?

One way to handle large numbers is to use "floating-point" numbers. Floating-point representation is what BASIC uses to handle single-precision and double-precision numbers. Floating-point operations are rather complex, however, and beyond the scope of these lessons.

There's no reason we can't handle any size number in the Z-80. We may have to string the numbers together as a series of bytes, but we can easily handle 4-byte or 8-byte numbers.

Look at Figure LESS11-1. In this figure, we have a 4-byte number. In four bytes, we can represent 2 to the 32nd power or about 4,295,000,000. That's not an unreasonable number range to work with, even for Federal budget deficits. As a matter of fact, we can get more **precision** than we get in single-precision BASIC. Note that I said more precision, which essentially means more digits; we still don't have the range of BASIC variables which also allow **exponents** such as 1.234×10 to the 14th power ($1.234E+14$).

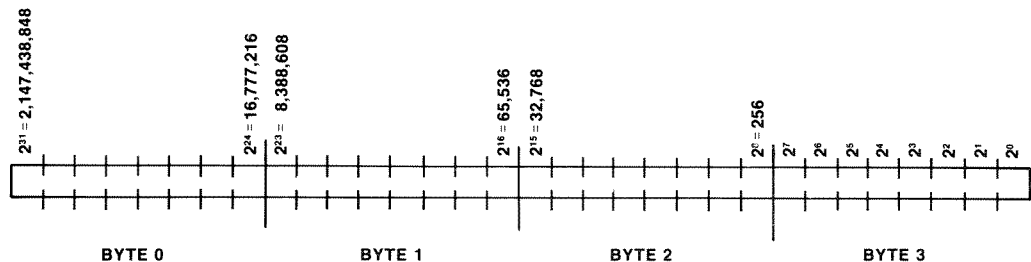


Figure LESS11-1. Four-Byte Multiple-Precision Number

If we want even more of a range, we can go to a larger number of bytes, but we'll consider 4 bytes here, for convenience.

The format of multiple-precision numbers is about the same as 8 or 16-bit numbers. The first bit may or may not be a sign bit, depending upon whether you're working with absolute or two's complement numbers. The only real difference is that the number is spread out among several bytes, and that adds, subtracts, and other operations have to be handled in 8-bit or 16-bit "chunks."

Suppose that we want to add the numbers shown in Figure LESS11-2. The two 4-byte numbers here represent 8,000,001 and 8,777,215. The first add adds the bytes 01 and FF, hexadecimal. The next add adds the next two bytes and any carry from the least significant byte. The next add adds the third bytes of the operands and any carry from the second add. Finally, the last add adds the fourth bytes of the operands and any carry from the third add.

11 Subtracts with Carry and Multiple Precision

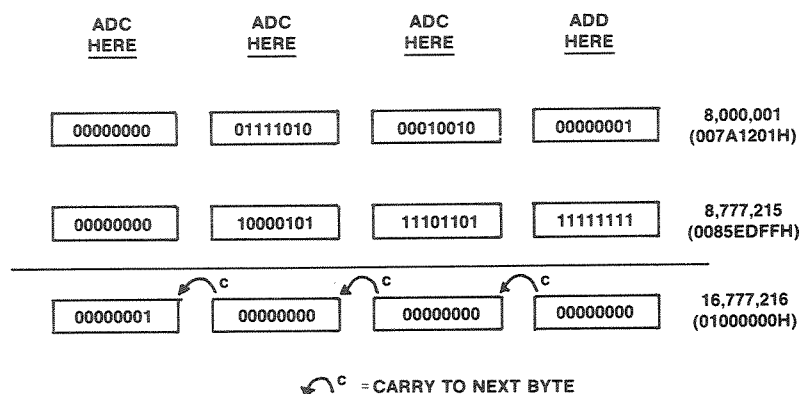


Figure LESS11-2. Adding Multiple-Precision Numbers

The first add is our old friend ADD, an 8-bit add. The remaining three adds are ADCs, or Adds with Carry so that any carry from the low order is added in.

Eight-Bit Add With Carry

The 8-bit ADCs operate very similarly to the standard 8-bit add. Another cpu register or memory location is added to the contents of the A register, with the result going into the A register. However, in addition to the second operand being added to the A register, the current state of the Carry flag is also added in. As the Carry flag may be set or reset, the add results in either the same sum as a normal ADD or a result that is one greater than the normal ADD.

Let's look at a program that performs the 4-byte multiple-precision add that we diagramed above. Enter the following source code, or use the existing source lines from the Lesson File.

```
100 ; ADC FOR MULTIPLE-PRECISION 4-BYTE ADDS
110 MPADDS    LD            HL,0B003H       ;point to op 1+3 bytes
120           LD            IX,0B007H       ;point to op 2+3 bytes
130           LD            B,4             ;loop counter
140           OR            A               ;clear carry
150 MPA010    LD            A,(HL)          ;get operand 1
160           ADC           A,(IX)          ;add in operand 2
170           LD            (HL),A         ;store result
180           DEC           HL              ;decrement op 1 pntr
190           DEC           IX              ;decrement op 2 pntr
200           DJNZ          MPA010         ;loop if not 4
210           END                          ;end
```

Assemble the source code and get an error-free assembly. Now trace the B000H area by

ZT B000

Modify the B000H area as follows: The four bytes at B000H through B003H hold the first operand. The four bytes at B004H through B007H hold the second operand. After the add is done, the result will replace the first operand at locations B000H through B003H. A suggested first set of operands is:

Operand 1	(B000H-B003H) 00H 80H 27H FFH	+8,398,847 decimal
Operand 2	(B004H-B007H) 83H A0H 7AH 11H	-2,086,634,991 decimal
Result	(B000H-B003H) 84H 20H A2H 10H	-2,078,236,144 decimal

Now execute the program by

ZX

and look at the results in the operand 1 area. Try making up several of your own operands.

The program works like this: HL is used as a pointer to the first operand area. Since we'll be adding bytes **from least significant byte to most significant**, HL is set to point to the last byte of the first operand.

IX is used as a pointer to the second operand area. It is initially set to point to the last byte of the second operand.

The B register is used as a loop counter for the 4 adds that will take place.

The loop starts at MPA010. Each time through the loop, HL and IX point to the next set of bytes in both operands.

The A register is loaded with the first operand byte by the register indirect of HL. The ADC instruction then adds the second operand byte. The result in A is then stored back in the first operand area. The ADC adds the two bytes, but also adds in any Carry from the previous add.

Important note: The only instruction affecting the Carry in the loop is the ADC, therefore the Carry always holds the Carry from the last ADC.

The Carry before the first add was set to 0 by the OR A instruction, which, as you will recall, sets the Carry to 0. The first ADC, then, uses a carry of 0.

Experiment with different 4-byte operands, and you'll get a good idea of how this multiple-precision add works.

Sixteen-Bit Add With Carry

We've already discussed the 16-bit ADD instruction that uses either HL, IX, or IY. There is a comparable 16-bit ADC instruction that uses HL only. The format of this ADC is

ADC HL,BC or ADC HL,DE or ADC HL,HL or ADC HL,SP

This add, like the ADD, works with the HL register pair and one other register pair, with the result going into HL. Like the 8-bit ADC, this ADC adds in any previous Carry.

Let's see how this ADC works. Delete lines 100 through 210 and enter the following source code, or use the Lesson File:

```

220 ; MULT PREC USING HL AND 4-BYTE OPERANDS
230 MPADDH    LD      BC,(B0006H)    ;get op 2
240          LD      HL,(B0002H)    ;get op 1
250          OR      A              ;clear carry
260          ADC     HL,BC          ;add two bytes
270          LD      (B0002H),HL    ;store
280          LD      BC,(B0004H)    ;get op 2
290          LD      HL,(B0000H)    ;get op 1
300          ADC     HL,BC          ;add two bytes
310          LD      (B0000H),HL    ;store
320          END                    ;end

```

Get an error-free assembly of this program, trace the B000H area, and then use the ZM command to store some data in the operand area. (Or use the same data as in the previous example.)

The operand area is identical to the operand area in the previous program. There is a problem with this add, however. Can you see what it is?

11 Subtracts with Carry and Multiple Precision

For the first operands, try these, and you will see the problem:

Operand 1 (B000H-B003H) 00H 80H 27H FFH

Operand 2 (B004H-B007H) 83H A0H 7AH 11H

Execute the program and look at the result.

We expected to see:

Result (B000H-B003H) 84H 20H A2H 10H

but we saw:

Result (B000H-B003H) 84H 20H A1H(!) 10H

Why? Unfortunately, the LD BC and the LD HL assume that the data to be loaded is in “reverse format,” with the least significant byte followed by the most significant byte. That’s not the way that we ordered the data — we went in the order most significant, next significant, next significant, and least significant.

If we want to use the 16-bit ADC for adding multiple-precision numbers, then, we had better order the data as shown in Figure LESS11-3. Other than that problem, using 16-bit ADCs for multiple-precision adds of 4 byte operands works fine. It is much faster than the 8-bit add method because it adds in 16-bit “chunks.” The size of the operands could be extended to any multiple of 2 bytes, or 16 bits.

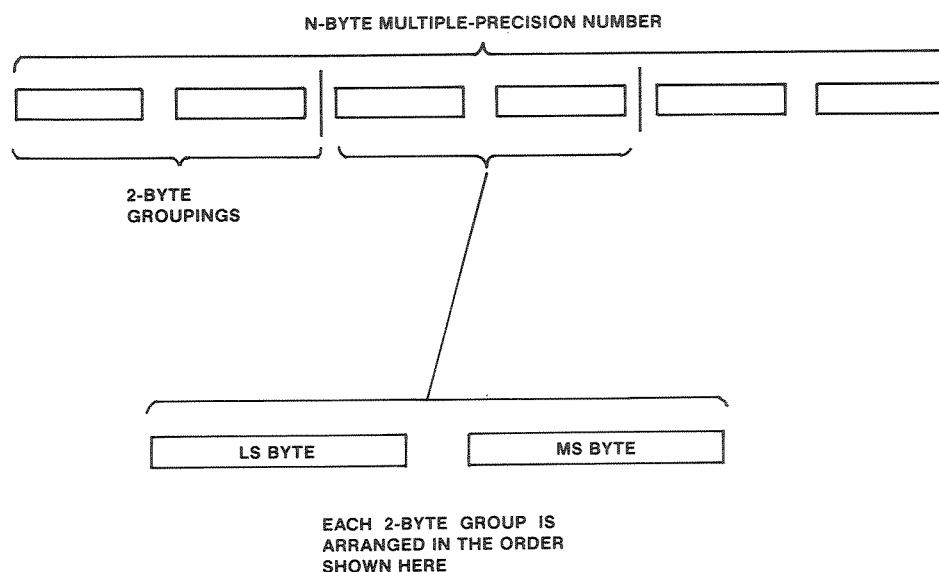


Figure LESS11-3. Sixteen-Byte Multiple Precision

Eight-Bit Subtracts With “Borrow”

The 8-bit subtract, SBC, is identical to the 8-bit ADC, except for the actual operation, of course. Again the addressing modes permitted are register-to-register, immediate, register indirect using HL, and indexed using either IX or IY.

In the SBC, any Carry from a lower order is actually a “Borrow,” but the borrow is termed a Carry for convenience.

To see that the subtract does work, delete lines 220 through 320, and enter the program below. It duplicates the earlier program, except that the ADC is replaced by an SBC. The operands are 4-byte operands in the B000H through B007H area as before.

330 ; SBC FOR MULTIPLE-PRECISION 4-BYTE SUBTRACTS

340 MPSUBS	LD	HL,OB003H	;point to operand 1+3
350	LD	IX,OB007H	;point to operand 2+3
360	LD	B,4	;loop counter
370	OR	A	;clear carry
380 MPSO10	LD	A,(HL)	;get operand 1
390	SBC	A,(IX)	;subtract operand 2
400	LD	(HL),A	;store result
410	DEC	HL	;decrement op 1 pntr
420	DEC	IX	;decrement op 2 pntr
430	DJNZ	MPSO10	;loop if not 4
440	END		;end

Assemble the program, trace the B000H area, and fill in some appropriate operands. Try this one as a start:

Operand 1	(B000H-B003H) 00H 80H 27H FFH	+8,398,847 decimal
Operand 2	(B004H-B007H) 83H A0H 7AH 11H	(-) -2,086,634,991 decimal
Result	(B000H-B003H) 7CH DFH ADH EEH	+2,095,033,838 decimal

Execute the program and look at the results in the B000H area. These operands cause a borrow from the next higher byte, and you can see how the SBC uses this borrow in the subtract of the current byte.

We'll leave it up to you to experiment with this multiple-precision subtract using other operands.

Sixteen-Bit Subtracts with Borrow

There is a 16-bit subtract with carry that is comparable to the 16-bit ADC. The 16-bit SBC is identical to the 16-bit ADC except for the actual operation. It uses HL and either BC, DE, HL, or SP. The second register pair is subtracted from the contents of HL, with the result going into HL. The possible combinations are:

SBC HL,BC or SBC HL,DE or SBC HL,HL or SBC HL,SP

We'll leave it up to you to enter and modify the 16-bit ADC program for multiple-precision subtraction. The instructions will be identical, except that an SBC HL,BC would be used in place of the ADC HL,BC.

Other Multiple-Precision Operations

You've seen in the above discussion how almost any size number can be handled for adds and subtracts by using ADC and SBC. But what about multiplies, divides, and other operations? Generally these are quite a bit more difficult. We'll be looking at some multiply and divide programs in a later lesson and we'll discuss some possibilities for multiple-precision operations in that lesson.

To Sum It All Up

To review what we've covered in this lesson:

- Multiple-precision numbers are numbers that use multiple bytes to provide a larger number range than is possible in 8 or 16 bits
- Multiple-precision numbers can be absolute or signed (two's complement) and resemble 8- or 16-bit numbers of these types
- Eight-bit adds with Carry operate similarly to the normal 8-bit adds, except that the current state of the Carry is added in
- Sixteen-bit adds with Carry use the HL register and another register pair, adding in the current Carry

11

Subtracts with Carry and Multiple Precision

- Eight-bit subtracts with Carry operate identically to an 8-bit ADC, except that a “borrow” is subtracted from the A register along with the second operand; the borrow is held in the Carry flag
- Sixteen-bit subtracts with Carry use the same register formats as the 16-bit ADC, using HL as an “accumulator” and another register pair as the operand to be subtracted from the HL register, along with the current state of the Carry

For Further Study

Look at the Flag operations for all 8- and 16-bit subtracts (Appendix V)

Lesson 12

How to Move a Block Away

Load LESS12 from disk.

In previous lessons we talked about how we could set up assembly-language loops by conditional branching using the JP or JR instructions. In this lesson we'll see how a loop can be used to move a block of data from one part of memory to another. We'll then look at a "built-in" loop in one instruction, using the Z-80 "block move" instructions.

First of all, let's answer the question: Why do we want to move data from one part of memory to another? Just to get clear in your mind what happens in this type of operation, look at Figure LESS12-1. This shows a "block move" from one part of RAM to another part of RAM. The block size may be any number of bytes, from 1 byte up to thousands of bytes.

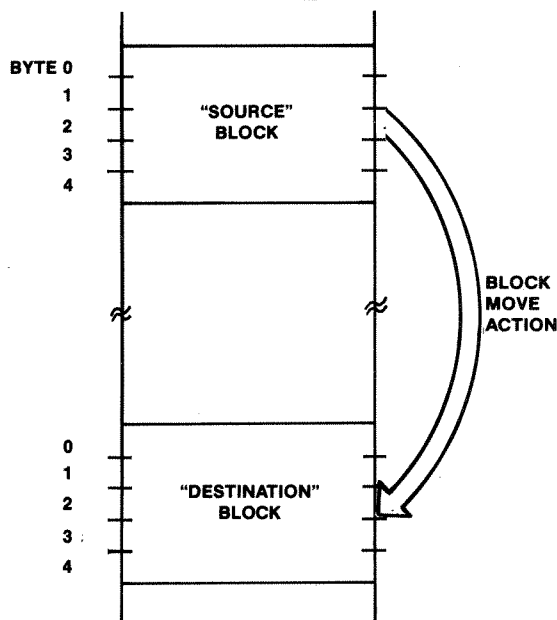


Figure LESS12-1. Block, Move Action

Block moves are used all the time in many different types of assembly-language operations. Some uses are:

- Shuffling blocks of data on "sorts" to arrange data in sequence according to some defined field in a record.
- "Scrolling" the screen up or down. (I'll bet you thought this was a hardware function!)
- Initializing "tables" or lists of data.
- Moving blocks of text for "inserts," "swaps," or deletions for word processing programs, such as Scripsit.

Block Move Number 1

Let's use some of the techniques we used in the preceding chapters to write a block move loop. Enter lines 100 through 300 of the following code, and assemble, or use the Lesson File.

12

How to Move a Block Away

100 ;NOT BAD FOR A HACKER

110	MOVBH	LD	HL,OBOOOH	;start of "source"
120		LD	DE,OBO10H	;start of "destination"
130		LD	B,16	;number of bytes
140	LOOP1	LD	A,(HL)	;get source byte
150		LD	(DE),A	;store in destination
160		INC	HL	;bump source
170		INC	DE	;bump destination
180		DEC	B	;count down to 0
190		JR	NZ,LOOP1	;continue if not done
200		END		;end

Got it? Now assemble the program and check for errors.

You're now ready to run. But first, trace memory locations B000H through B01FH by entering

ZT B000

After the ZT, you should see the B000H area displayed in the experiment area. ALT has filled this area with an easily recognizable pattern from the Lesson File, as shown in Figure LESS12-2.

LOCN	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
B000	11	22	33	44	55	66	77	88	99	00	11	22	33	44	55	66
B010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure LESS12-2. Experiment Area Data Pattern

Now do a

ZS 5000

ZX

to set the speed to moderate and to execute the program from the start.

You should see the "source" bytes replace the destination bytes one by one. At the end of the time, the entire "destination line" has been filled with the "source line" contents. The source line contents remains unchanged. The source block has been copied into the destination area.

As you execute the program slowly, watch the pointers change in HL and DE, and the byte count decrement down to 0 in B.

Changing the Number of Bytes Moved

By changing the value loaded into B in line 130, you can change the number of bytes moved. Try experimenting with different byte counts in B and data in the B000H area (use the ZM command to modify the B000H area, and the Delete and Insert commands to change line 130).

Changing the Source and Destination Pointers

Do you see how the source and destination areas are defined? The source "pointer" is loaded into the HL register pair. The destination pointer is loaded into the DE register pair. Try experimenting with different source and destination addresses in HL and DE, but remember to use the "experiment" memory in B000H through B03FH, otherwise the ALT will prompt you that you've gone beyond your bounds again!

How the Program Works

If you can't figure out what MOVEBH does, let's go through it step-by-step. The instruction in line 110 loads the HL register pair with the "source" address. The instruction in line 120 loads the DE register pair with the "destination" address. The next instruction loads the B register with the number of bytes to be moved. These three instructions are the "initialization" of the loop following.

The instructions from line 140 through line 190 are the actual loop. A byte is picked up from the source area by a load of A, using HL as a "pointer." This byte is stored into the destination area, using DE as a pointer. Both pointers in HL and DE are then incremented by one count, to point to the next byte. The decrement then subtracts one from the number of bytes in the B register.

If the count in B has not been reduced down to 0, the JR goes back to location LOOP1 to do the operation another time.

Eventually the count in the B register goes down to 0, and the END instruction causes a return to the ALT monitor.

A Better Block Move

Programmers are always trying to perfect their "code" to make it run better. Can we perfect this code? No? All right, let's move on to Lesson 13...

Wait a minute! We **can** perfect this code. Let me ask you a question first. How many bytes can be moved at one time with the MOVEBH program?

If we load 100 in the B register, we'd move 100 bytes. Loading 200 in the B register would move 200 bytes. Loading 255 in the B register would move 255 bytes. What if we loaded the B register with 0?

If we loaded the B register with 0, the DEC instruction would subtract 1 from the B register as follows

```
00000000
-00000001
-----
11111111
```

The result would be all ones, or 255! The count of 255 would then cause 255 "iterations" through the loop until 0 was reached. Actually, then, putting 0 in the B register causes 256 bytes to be transferred.

The maximum number of bytes we can transfer is therefore 256 bytes. We could use this program several times to transfer larger numbers of bytes, but how about a new program that will transfer thousands of bytes. Any ideas?

If you've scrolled ahead, you know we have one. Delete lines 100 through 200 to see the following code:

```
210 ;A BETTER MOVE BLOCK
220 MOVBB LD HL,0B000H ;start of "source"
230 LD DE,0B010H ;start of "destination"
240 LD BC,16 ;number of bytes
250 LOOP2 LD A,(HL) ;get source byte
260 LD (DE),A ;store in destination
270 INC HL ;bump source
280 INC DE ;bump destination
290 DEC BC ;count down to 0
300 LD A,B ;test bc
310 OR C
320 JR NZ,LOOP2 ;continue if not done
330 END ;out
```

This program is very much like the first, with a few exceptions.

12 How to Move a Block Away

The BC register pair is used to hold the number of bytes to be moved. We can move 1 through 65,536 bytes by specifying a count of 1 through 65,535 in the LD BC instruction (0 is 65,536).

Also, since the DEC BC does **not affect the flags**, we have to be somewhat devious to test BC to see if the count has been decremented down to 0. The test is made by loading the A register with the contents of the B register, and then ORing in the contents of the C register. The only way the Z condition will be true is if the count has decremented down to 0 (B=0, C=0).

Assemble and run MOVBB, using the same procedure as in the first program. Of course, here we're only moving 16 bytes, but we could move thousands by changing the pointers and number of bytes in BC.

Put new data in the B000 AREA.

As you execute the program slowly, watch the pointers change in HL and DE, and the byte count decrement down to 0 in BC.

Good, Better, Best . . .

All right, are you ready for the all-time fastest version of move block, one that is only 1/3rd the size of MOVBB?

Delete lines 210 through 330, and you'll see it:

340	: MOVE BLOCK BY LDIR		
350	MOVBL	LD	HL,0B000H ;start of "source"
360		LD	DE,0B010H ;start of "destination"
370		LD	BC,16 ;number of bytes
380	LOOP3	LDIR	;move block!
390		END	;end

Run the program after assembling, using the same commands as in the other two programs. Neat eh?

Did you notice in running this program that the data appeared to be transferred instantaneously? That's because the LDIR is a single instruction, and the ALT program pauses between each instruction during speed control; the LDIR performed the 16 iterations of the loop all by itself.

The LDIR replaces the loop of MOVBB with one instruction. It assumes that the HL register pair points to the "source" block starting address, that the DE register pair points to the "destination" block starting address, and that the BC register pair points to the number of bytes to be moved, as shown in Figure LESS12-3.

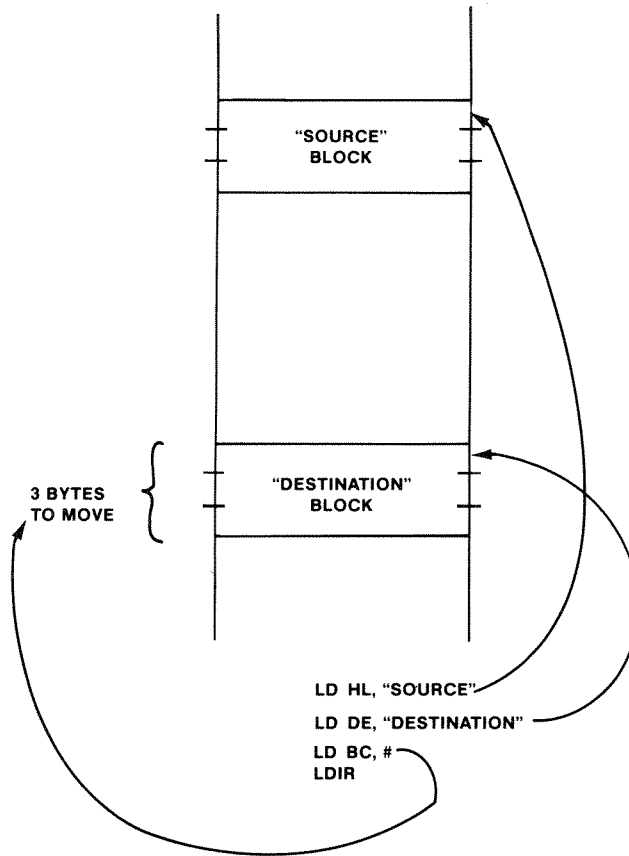


Figure LESS12-3. LDIR Set Up

Experiment, using different starting addresses and byte counts with the LDIR. It always operates the same as the loop in MOVBB.

How to Get in Trouble with an LDIR

What could be simpler — one instruction for a block move. There is a minor problem. . . Delete lines 340 through 390, and you'll see a program that'll show you what the problem is:

```

400 ; MOVE BLOCK BY LDIR
410 MOVBX    LD      HL,0B000H    ;start of "source"
420          LD      DE,0B001H    ;start of "destination"
430          LD      BC,16        ;number of bytes
440 LOOP4    LDIR                     ;move block!
450          END                      ;end

```

Assemble and run the program as before. Did you notice anything strange?

It appears that instead of moving the data in locations B000H through B00FH to the area one byte higher, B001H through B020H, the value in B000H was repeated throughout the entire destination block!

This is a problem that occurs when the blocks are **overlapping**. If you'll give it some thought, you'll see what happened. The byte at location B000H was first transferred to location B001H. The byte at B001H was then transferred to location B002H, but this byte was the byte from location B000H! This "chaining" operation was repeated for all 16 bytes and resulted in replication of the data from the byte at B000H.

12

How to Move a Block Away

The moral of this story is Be Careful! If the source block is followed by the destination block, and they overlap, part of the data will be replicated! Too bad there isn't an instruction that will work in reverse to avoid that problem.

The LDDR

Turns out there is! The LDDR works very similarly to the LDIR, except in reverse. The HL and DE registers point to the **end** of the blocks to be moved while the BC register contains the number of bytes to be moved as before. Delete lines 400 through 450, and you'll see what I mean:

```
460 ; MOVE BLOCK BY LDDR
470 MOVBD    LD      HL,0B00FH    ;end of "source"
480          LD      DE,0B010H    ;end of "destination"
490          LD      BC,16        ;number of bytes
500 LOOP5    LDDR           ;move block!
510          END                ;end
```

Assemble and execute the program as before. Note that this time the move was done successfully. The data in B000H through B00FH was moved to B001H through B010H (B000H was unchanged). Look at the HL and DE registers at the end of the move; they point to AFFFH and B000H, respectively (the pointers were adjusted to these locations, but the byte count "ran out" after 16 iterations). The byte count in BC was adjusted down to 0.

To Sum It All Up

To recap the LDIR and LDDR, then:

- Use LDIR or LDDR in place of a loop of other instructions anytime you want to move data
- Always set HL to the source start and DE to the destination start for LDIR or to the source end and destination end for LDDR
- The BC register always contains the number of bytes to be moved
- If an LDIR is used, the data will be transferred from beginning to end; if an LDDR is used, the data will be transferred from end to beginning
- If the blocks are overlapping, use LDIR when the source block starts **higher** in memory than the destination block or LDDR when the source block starts **lower** in memory than the destination block

For Further Study

LDI, LDD instructions (Appendix V)

Lesson 13

Table Techniques Part I

Load LESS13 from disk.

“Tables” are one of the most important **data structures** in assembly-language programming. A table is a list of data, arranged in convenient form for “access.” Tables can be constructed at “assembly time” or can be constructed “dynamically” during program execution.

A List of Data

The simplest form of a table is simply a “one-dimensional” list of data, similar to a one-dimensional BASIC array. Let’s build two versions of a data list, one with numeric data and one with “alphanumeric” (text) data.

A Numeric Table Lookup

Suppose that we wanted to convert from degrees Centigrade to degrees Farenheit. One way to do it would be to use the formula:

$$F=(9/5)*C+32$$

where C is the temperature in degrees Centigrade and F is the temperature in degrees Farenheit. We could do a multiply, a divide, and an add, but another alternative would be to use a “look-up” table of Farenheit values. This table would appear as in Figure LESS13-1.

CENTIGRADE	
0	32
1	34
2	36
3	37
4	39
5	41
6	43
7	45
8	46
...	
95	203
96	205
97	207
98	208
99	210
100	212

TABLE ENTRIES ARE CORRESPONDING DEGREES FARENHEIT

Figure LESS13-1. Centigrade to Farenheit Table

The table is “accessed” by using the number of Centigrade degrees as an “index” value. The table starts with 0 degrees Centigrade and ends with 100 degrees Centigrade. To find the Farenheit degrees for 20 degrees Centigrade, for example, you’d look up the 21st “entry” in the table.

How would you construct such a table using assembly language? To see the answer, look at the first portion of the Lesson File, which gives the first 20 entries of such a table:

13

Table Techniques Part I

100 ; LOOKUP TABLE FOR CENTIGRADE TO FARENHEIT CONVERSION
110 ; ONE ENTRY FOR EVERY DEGREE CENTIGRADE

120 CTOFTB	DEFB	32	;0 degrees C
130	DEFB	34	;1
140	DEFB	36	;2
150	DEFB	37	;3
160	DEFB	39	;4
170	DEFB	41	;5
180	DEFB	43	;6
190	DEFB	45	;7
200	DEFB	46	;8
210	DEFB	48	;9
220	DEFB	50	;10
230	DEFB	52	;11
240	DEFB	54	;12
250	DEFB	55	;13
260	DEFB	57	;14
270	DEFB	59	;15
280	DEFB	61	;16
290	DEFB	63	;17
300	DEFB	64	;18
310	DEFB	66	;19

Assemble this source code and see what you get.

Ok? You should have seen a list of 20 bytes, ranging from 32 (20H) through 66 (42H).

The DEFB Pseudo-Op

The DEFB is an assembler “pseudo-op” that does not generate an instruction, but generates “data” instead. Unlike BASIC, though, we have to be careful where we put the data. In BASIC the DATA statements are simply “in-line” with other BASIC statements. In assembly-language we can put data anywhere, but have to make certain that the code jumps around them.

Hexadecimal values for DEFB that start with an A through F must have a “leading” 0 for the DEFB.

You can see from the above code that you can use a label on a data area as well as an instruction. This allows you to symbolically reference the data, which, in this example, we’ve called CTOFTB, or “C to F Table.”

Each “entry” in this table is one byte long. Given a temperature reading in degrees Centigrade, we can easily find the equivalent degrees Farenheit by a “table lookup.”

To the code above add the following source code, or use the existing source code from the Lesson File:

320 ; TABLE LOOKUP FOR C TO F CONVERSION

330 TABLOK	LD	HL,CTOFTB	;load table start
340	LD	A,(0B000H)	;get degrees C
350	LD	C,A	;now in C
360	LD	B,0	;now in BC
370	ADD	HL,BC	;HL now points to data
380	LD	A,(HL)	;get degrees F
390	LD	(0B001H),A	;put into memory
400	END		;end

Assemble the source code and get an error-free assembly.

This program takes a value in degrees Centigrade from memory location B000H, uses it as an index value, and “looks up” the corresponding degrees Farenheit in the CTOFTB.

HL is used as an indirect pointer, and is loaded with “CTOFTB.” This symbol is the same as any other symbol used with an instruction. It is the symbolic name of the location, in this case of data. The corresponding address of CTOFTB is put into the instruction as immediate data, as you can see.

The A register is then loaded with the degrees Centigrade value from location B000H. This value is transferred to the C register, and the B register is cleared. At this point, the BC register pair now holds 00XXH, where XX is the degrees Centigrade value. This value is the “index value” for the table lookup.

Next, the index value in BC is added to the start of the table value in HL. The result in HL points to the location in the table where the Farenheit byte will be found.

The A register is then loaded with this value, and the value is then stored in location B001H.

Execute the program after first setting “slow mode,” tracing the B000H area, and putting a Centigrade value (00H — 13H) in location B000H with the ZM command. You must execute from the hex location of “TABLOK” by doing a ZX MMMM. The result will be in location B001H. (If you use values other than 00H — 13H, you will get a “NO DATA” or other error message.)

Entries of More Than One Byte

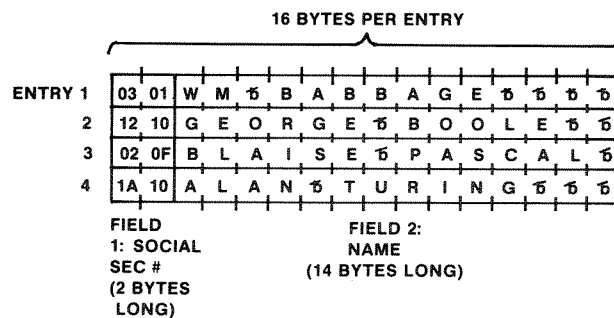
The table above was one of the simplest tables we could work with. Let’s look at a more complicated table, one that uses “entries” greater than one byte and also uses several “fields.”

Delete lines 100 through 400 and enter the program below or use the Lesson File:

```

410 ; TABLE WITH LARGE ENTRIES
420 EMPTAB      DEFW      0103H
430             DEFM      'WM BABBAGE '
440             DEFW      1012H
450             DEFM      'GEORGE BOOLE '
460             DEFW      0F02H
470             DEFM      'BLAISE PASCAL '
480             DEFW      101AH
490             DEFM      'ALAN TURING '
    
```

This table has four entries, but, of course, it could have many more. The table is shown in Figure LESS13-2.



␣ = BLANK

Figure LESS13-2. Table with Multiple-Byte Entries

13

Table Techniques Part I

Each entry is made up of two “fields.” A field is a subdivision of a “record,” as we saw in an earlier lesson. The second field holds the name of the computer pioneer, and the first field holds his Social Security Number (they used shorter numbers in those days, of course). Each field is a “fixed length”; the second field is 14 bytes long, and the first field is 2 bytes long. The total length of each entry is 16 bytes.

This concept of entries in a table with fields within the entry could be expanded to tables with hundreds of entries and with many fields. For example, you might have a table of employees (for the ACME Z-80 Programming School, or other business) that had one field for an employee name, another for an employee number, another for marital status, and so on. Each entry in the table might be a hundred bytes long or so.

Why did we make the fields “fixed length” above? In fact, we could have used a “variable length” field, which would make the entries variable length also. Fixed-length entries are easier to work with, even though they do take up more space.

The DEFW Pseudo-Op

This is the first time we’ve used the DEFW pseudo-op. You’ll recall that a pseudo-op is an instruction to the assembler that tells it that something other than an instruction mnemonic is coming.

The DEFW is used to build data, just as the DEFB was used in the previous example. In this case the data for the DEFW represents a 16-bit value. Instead of one byte being generated, as in the case of the DEFB, two bytes will be produced. Those two bytes will be in “reverse format,” as we’ve seen for other Z-80 data such as immediate data in instructions.

Hexadecimal values that start with an A through F must have a “leading” 0 for DEFW.

The DEFM Pseudo-Op

The DEFM pseudo-op is also new. The DEFM generates ASCII bytes. ASCII, of course, is a special code used to represent text data. All assemblers have such a pseudo-op to allow the programmer to easily construct messages and other text data.

The DEFM generates one ASCII byte for each character enclosed within the quotes. Only the first four bytes will be shown on the assembly listing.

To see how DEFM and DEFW work, assemble the source lines above and get an error-free assembly. After assembly, first Trace the table constructed by entering

ZT XXXX

where XXXX is the location of the table from the assembly listing. You’ll see the first part of the table in the Memory Trace area.

To see the ASCII equivalent, use the “T” option (for “Text”) for the Memory Trace. Enter

ZTT XXXX or simply **ZTT** alone

and the memory area will be displayed in ASCII format. You should now see the names in field 1 of the table, but the DEFW data in field 2 will appear as periods, as the values are not legitimate ASCII characters.

Switch back and forth between ZT and ZTT and you’ll see how both types of data are stored after assembly. Use ZT+ and ZT- to “scroll” forward or backwards through the data area.

Accessing Multiple-Byte Table Entries

We can “scan” a table such as the one above just about as easily as we did in the one-byte per entry table case. Scanning means going through the table one entry at a time and trying to find a given entry.

In this case, though, we have to adjust the table pointers by 16 to get to the next entry.

Suppose that we are looking for the Social Security Number “key” held in memory locations B000H and B001H. We can use the following program to scan the 4-entry EMPTAB table. Assemble to get the program below:

```

500 ; TABLE LOOKUP FOR FINDING SOCIAL SECURITY NUMBER
510 SSNLOK    LD      HL,EMPTAB    ;load start of table
520          LD      DE,16        ;entry size
530          LD      B,4          ;for 4 entries
540 SSNO10    LD      A,(OB000H)    ;get first byte of #
550          CP      (HL)          ;compare
560          JR      NZ,SSNO20     ;go if not equal
570          INC     HL            ;point to next byte
580          LD      A,(OB001H)    ;get 2nd byte of #
590          SUB     (HL)          ;compare
600          DEC     HL            ;adjust fnd or not fnd
610          JR      Z,SSNO30      ;go if “found”
620 SSNO20    ADD     HL,DE        ;point to next entry
630          DJNZ    SSNO10        ;go again if not 4
640          LD      A,OFFH        ;“not found” flag
650 SSNO30    LD      (OB002H),A   ;store flag
660          END

```

This is a fairly complicated program, so we’ll explain carefully how it works. The flowchart is shown in Figure LESS13-3.

13 Table Techniques Part I

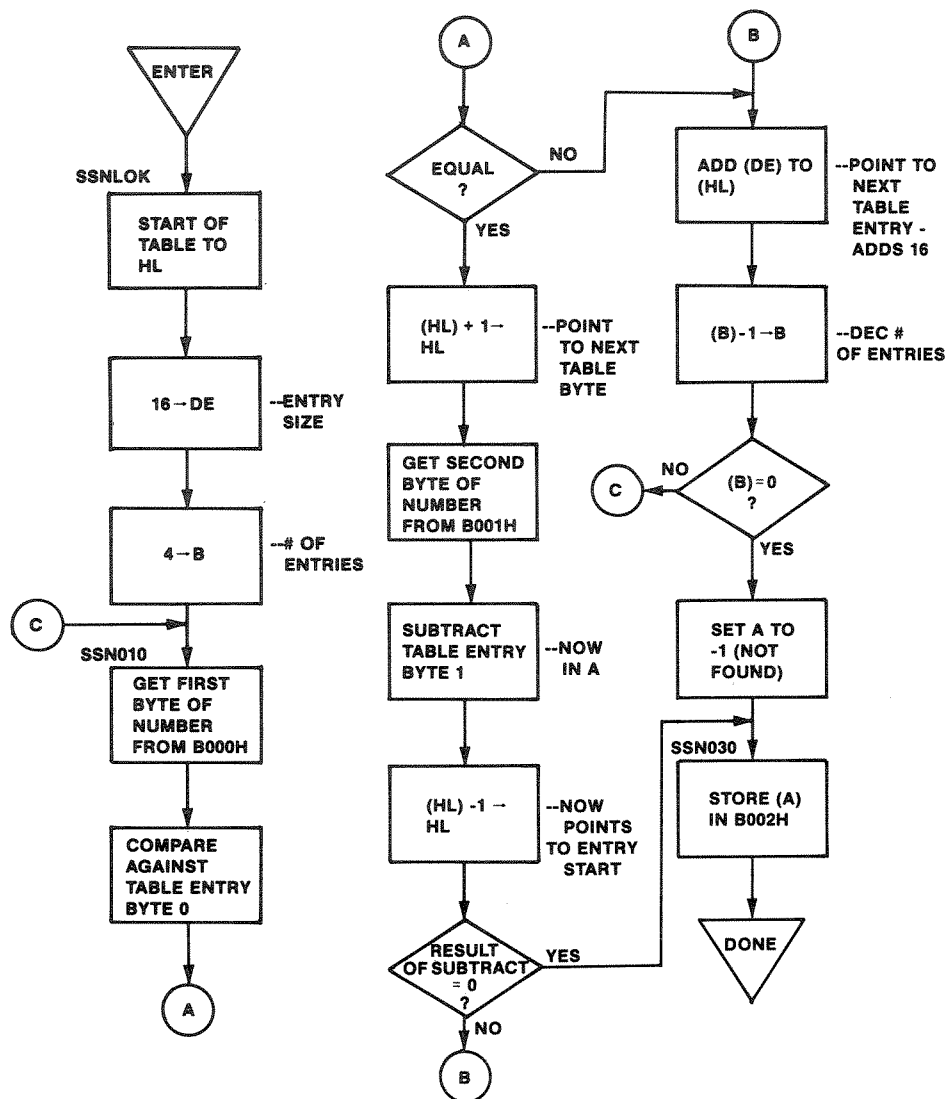


Figure LESS13-3. SSNLOK Flowchart

First of all, HL is loaded with the address of the table. HL will point to each entry in the table in turn. Next, DE is loaded with 16. DE will be added to HL so that HL will point to the next entry in the table for each of 4 times through a loop. Next, B is loaded with 4. Since there are 4 entries, we'll have to go through a loop 4 times to compare each entry.

The loop starts at SSN010. Each time through the loop, the following actions occur:

- The first byte of the Social Security number is loaded into A from memory location B000H.
- This value is compared to the table value pointed to by HL. This would be the first byte of the "current" entry.
- If they are not equal (NZ), a jump is made to SSN020 to adjust the HL pointer to the next entry.
- HL is then incremented by one to point to the next byte in the table entry.
- The next byte of the number in location B001H is loaded into A.

- A subtract of (HL) is done. This subtracts the next byte of the table entry from A, puts the result (zero or non-zero) in A, and sets the flags.
- HL is adjusted back to the start of the table entry by decrementing by one.
- If the second bytes are equal (Z), the number has been found. In this case a jump is made to SSN030.
- If the first or second bytes are not equal, the ADD HL,DE adjusts HL by 16 to point to the next table entry. A DJNZ then loops back to SSN010 for the next comparison. If the count in B is decremented down to 0 by the DJNZ, the loop is done, A is loaded with -1, and a jump is made to SSN030.
- SSN030 stores the value in A in location B002H. This value is 0 (from the subtract) if the number is found, or -1 if the number isn't found after 4 compares.
- An important note: If the number is found, **HL points to the table entry for the number**. This is the most important result of this table "scan."

Enter a social security number into B000H and B001H to correspond to one of the table entries. Don't forget to use "reverse format." Now execute the program at low speed and watch the HL register.

At the end of the program, A should contain the "found"/"not found" flag and HL should point to the table entry if it was found.

We'll look at more table techniques in the next lesson.

To Sum It All Up

To review what we've learned in this lesson:

- A simple table might be a list of one-byte entries
- A table can be accessed by using an "index value." The index corresponds to one parameter, and the entry corresponding to the index is another related value
- The DEFB generates a one-byte data value
- Data can be labeled with symbolic names, just as instructions can be labeled
- Tables with multiple-byte entries are common
- Entries in tables may be subdivided into fields
- Tables may have "fixed-length" or "variable-length" entries
- The DEFW generates two bytes of data in standard Z-80 16-bit format
- The DEFM generates a string of ASCII characters
- "Scanning" a table means that a search of the table is performed; the search is for a specific entry

For Further Study

DEFB, DEFW — private study

DEFW and ASCII codes — private study



Lesson 14

Table Techniques Part II

Load LESS14 from disk.

Let's take another look at the program from Lesson 13. A modified version is shown below as it appears in the Lesson File for Lesson 14. Enter the source lines below, or use the Lesson File. Assemble and execute just as you did in the previous lesson, putting a "search" value in locations B000H, B001H, and looking for a "found"/"not found" flag in location B002H on completion.

```
100 ; TABLE LOOKUP FOR FINDING SOCIAL SECURITY NUMBER
110 SSNLOK      LD      IX,EMPTAB      ;load start of table
120             LD      DE,16          ;entry size
130             LD      B,4            ;for 4 entries
140 SSN010      LD      A,(B000H)      ;get first byte of #
150             CP      (IX)           ;compare
160             JR      NZ,SSN020      ;go if not equal
170             LD      A,(B001H)      ;get 2nd byte of #
180             SUB     (IX+1)          ;compare
190             JR      Z,SSN030       ;go if "found"
200 SSN020      ADD     IX,DE           ;point to next entry
210             DJNZ    SSN010         ;go again if not 4
220             LD      A,OFFH         ;"not found" flag
230 SSN030      LD      (B002H),A      ;store flag
240             END                    ;end
250 EMPTAB      DEFW     0103H
260             DEFM     'WM BABBAGE   '
270             DEFW     1012H
280             DEFM     'GEORGE BOOLE '
290             DEFW     0F02H
300             DEFM     'BLAISE PASCAL '
310             DEFW     101AH
320             DEFM     'ALAN TURING  '
330             END
```

Compare this program with the version from Lesson 13 and see if you can find the differences.

The biggest difference is that the IX register is used in place of the HL register. Where in the previous version HL pointed to each byte of the table entry, in this case IX is not adjusted by an INC or DEC.

Indexed Addressing

The type of addressing mode we're using here is called "indexed addressing." We've used the IX and IY registers previously in this course, but only as "register pointers" in identical fashion to using HL. In this example, IX is used as a "base index register," pointing to the start of an area. The "+1" in the IX+1 is a "displacement" value that is added to the IX pointer value to find the actual location pointed to.

Figure LESS14-1 shows what we mean. Here IX points to location 8000H. Doing an LD A in indexed addressing mode, however, allows us to load A not only with the contents of location 8000H, but with any memory byte from 8000H-128 to 8000H+127.

14 Table Techniques Part II

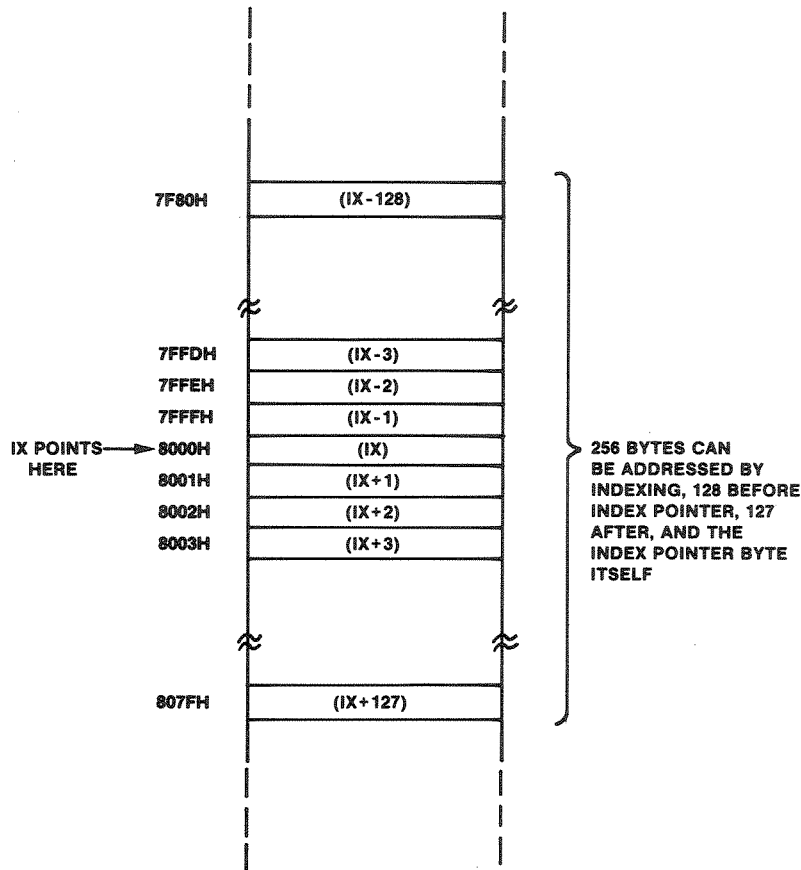


Figure LESS14-1. Indexed Addressing

Delete source lines 100 through 330 and enter the following source lines, and we'll show you what we mean:

```

340 ; INDEXING EXAMPLE
350 INDSTR LD IX,B0003H ;point to base
360 LD A,(IX-3) ;get -3 byte
370 LD B,(IX+3) ;get +3 byte
380 LD (IX+3),A ;swap
390 LD (IX-3),B
400 LD A,(IX-2) ;get -2 byte
410 LD B,(IX+2) ;get +2 byte
420 LD (IX+2),A ;swap
430 LD (IX-2),B
440 LD A,(IX-1) ;get -1 byte
450 LD B,(IX+1) ;get +1 byte
460 LD (IX+1),A ;swap
470 LD (IX-1),B
480 END ;end
  
```

This isn't an especially profound program, but it does illustrate how indexing works. Assemble the program and execute it while tracing the B000H area. The data in B000H through B002H will be put into locations B006H through B004H, respectively (reverse order), by indexing the B000H area. Note that IX never changes. It points to the base address of B003H.

Now look at the machine-language instructions assembled for the indexed instructions. The first and

sometimes second bytes of indexed instructions are the “opcode.” The last byte is always a “displacement.”

The displacement is a two’s complement number in the range of -128 through +127, very similar to the displacement used in the relative jump instructions or the DJNZ instruction. Look at the displacement in LD A,(IX-2), for example. The displacement byte here is FEH, which is a binary 11111110, or a -2 in two’s complement form.

The “effective address” for an indexed instruction is found by taking the contents of the index register and adding the **sign extended** displacement to it. The result is the address used in the instruction.

In the case of this instruction, for example, the effective address would be

```

10110000 00000011 (B003H in IX)
11111111 11111110 (-2 in displ)
-----
10110000 00000001 (B001H = EA)

```

“EA” stands for “effective address” in this computation. This effective address of B001H would be used in the LD A, so in effect:

LD A,(IX-2) = LD A,(B001H)

in this case.

If you look at the other indexed type instructions, you can see that they also have displacement bytes that duplicate the value in the source line.

Of course, we’re only showing how the effective address is computed here. You don’t have to worry about the actual computation; the assembler will automatically take care of it for you. All you have to do is to establish the IX or IY register at some base value and then use the proper displacement in parentheses, such as (IX+23), (IX-67), (IY+23), or (IY-12).

Table Operations Using Indexing

Indexing is especially useful in working with tables. Suppose that we have a table that contains entries for a simple inventory system for a computer manufacturer. The table and entries are shown in Figure LESS14-2.

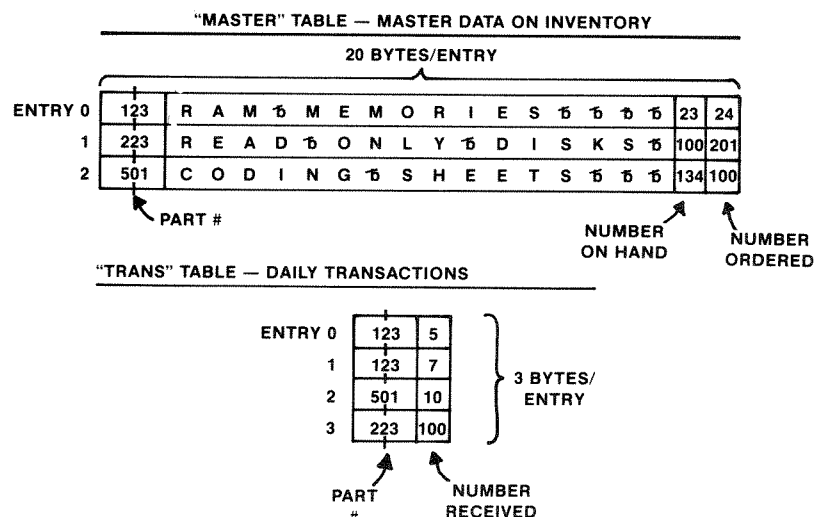


Figure LESS14-2. Inventory Table Example

14 Table Techniques Part II

Each entry in the “master” table is made up of four fields.

Field number 1 is the part number, from 0000 through 9999, in 2 bytes.

Field number 2 is the description of the part. This is a fixed-length field of 16 characters.

Field number 3 is the number “on hand,” the number actually at the manufacturer, in one byte.

Field number 4 is the number on order, the number of parts that haven’t yet come in, in one byte.

Each entry in the table is therefore 20 bytes long.

Each entry in the “transaction” table, a second table, is made up of two fields, a part number (two bytes) and the number received, in 1 byte. Each entry is therefore 3 bytes long.

We want to write a program that will adjust both the number on order and the number on hand in the “master table” to reflect the number of parts received from the “transaction” table. One way to implement it is shown below. Delete lines 340 through 480 and enter the code below, or use the Lesson File.

```

490 ; INVENTORY PROGRAM USING INDEXING
500 ; UPDATES MASTER TABLE WITH DATA FROM TRANSACTION TABLE
510 INVSTR      LD          IY,TRANS      ;address of transaction
520             LD          B,4           ;number of transactions
530 INVO10      LD          IX,MASTER     ;address of master
540 INVO20      LD          L,(IY)        ;get part #
550             LD          H,(IY+1)
560             LD          E,(IX)        ;get table part #
570             LD          D,(IX+1)
580             OR          A            ;clear carry
590             SBC         HL,DE         ;test for part #
600             JR          NZ,INVO30     ;go if not equal
610             LD          A,(IY+2)      ;get “# received”
620             ADD         A,(IX+18)     ;add on hand, recvd
630             LD          (IX+18),A     ;store new on hand
640             LD          A,(IX+19)     ;get # ordered
650             SUB         (IY+2)        ;# ordered — # recvd
660             LD          (IX+19),A     ;store
670             JR          INVO40        ;go for next part
680 INVO30      LD          DE,20         ;for master table
690             ADD         IX,DE         ;point to next
700             JR          INVO20        ;get next from master
710 INVO40      LD          DE,3          ;for trans table
720             ADD         IY,DE         ;point to next
730             DJNZ        INVO10       ;go if not done
740 ENDLOC      JR          ENDLOC        ;done
750 MASTER      DEFW        123          ;part number 123
760             DEFM        'RAM MEMORIES '
770             DEFB        23
780             DEFB        34
790             DEFW        223          ;part number 223
800             DEFM        'READ ONLY DISKS '
810             DEFB        100
820             DEFB        201
830             DEFW        501          ;part number 501
840             DEFM        'CODING SHEETS '
850             DEFB        134
860             DEFB        100

```

870 TRANS	DEFW	123	;part number 123
880	DEFB	5	;5 received
890	DEFW	123	;part number 123
900	DEFB	7	;7 received
910	DEFW	501	;part number 501
920	DEFB	10	;10 received
930	DEFW	223	;part number 223
940	DEFB	100	;100 received
950	END		;end

The two tables are established in the program itself, one called MASTER and the other TRANS.

This program will call for everything you've learned thus far, but don't flinch! We'll follow it step by step, and you can run it in slow speed and observe the results.

A flowchart for the program is shown in Figure LESS14-3.

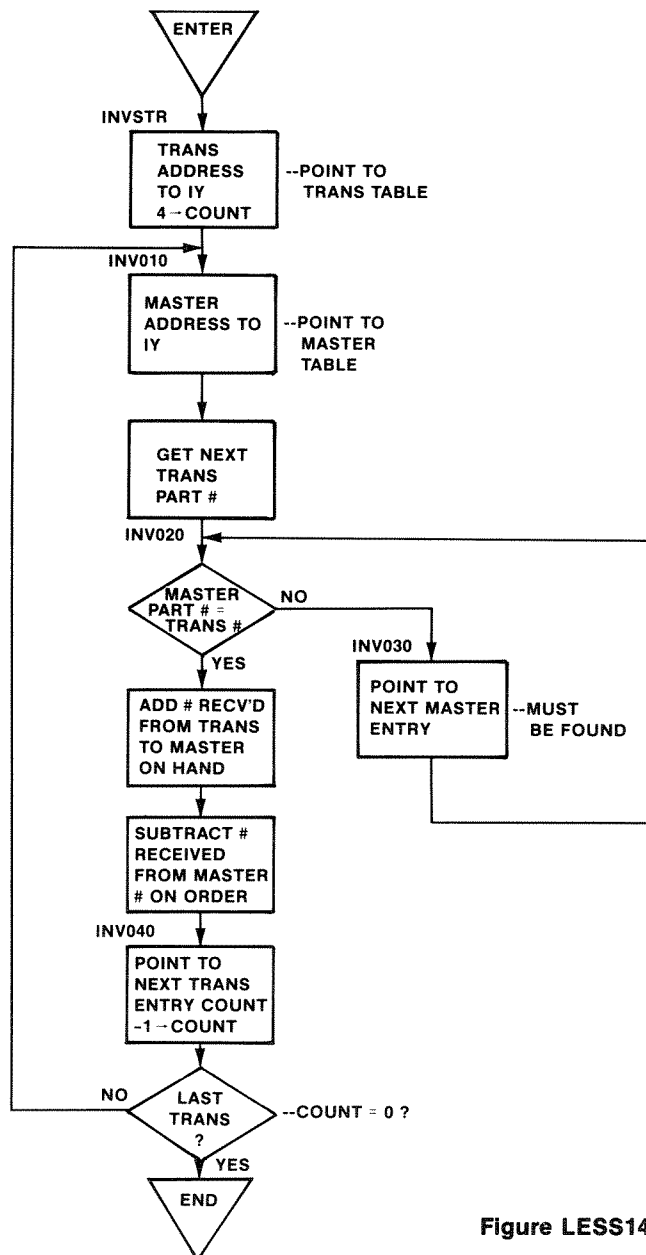


Figure LESS14-3. INVSTR Flowchart

14

Table Techniques Part II

IX is used in the program to point to the MASTER table. IY is used to point to the TRANS table. IY is used to go down the TRANS table one entry at a time. Each time one entry is obtained from TRANS, the MASTER table is “scanned” for the matching part number. **We’re assuming the part number will always be found, by the way. If it is not, what will happen?**

When the part number is found, the number received from the TRANS table is added to the number on hand, and the updated number on hand is put back into the MASTER table. The number received is then subtracted from the number on order and the result is put back into the number on order.

All through the program, IY points to TRANS and IX points to the master entries.

Here’s a detailed description:

IY is first loaded with the address of TRANS. Next, B is loaded with 4. There are 4 entries in TRANS, so we’ll have to make 4 updates to the MASTER table.

The “outer loop” starts at INV010. The outer loop is done 4 times, once for each entry in TRANS. The outer loop contains an “inner loop” which searches the MASTER table for the part number.

At INV010, the address of MASTER is loaded into IX. In other words, we’re starting again from the “top” of the MASTER table to look for the part number.

Next, the part number of the current TRANS entry is loaded into HL. Notice the method in which HL is loaded, by using IY indexing to load the next 2 bytes from TRANS.

The inner loop starts at INV020. At this point IY points to the current TRANS entry and IX points to the first entry in MASTER.

DE is loaded with the part number from the MASTER table. This part number is subtracted from the TRANS part number in HL. If they are not equal (NZ), INV030 is executed to add 20 to the IX register. This points to the next entry in MASTER.

If the part numbers are equal, the number received is added to the number on hand, and the number received is subtracted from the number ordered. The results are stored back in MASTER. The outer loop code at INV040 is then executed. This code increments the IY (TRANS) pointer by 3 so that it points to the next TRANS entry. DJNZ then decrements B and jumps back to INV010 for the next TRANS part number.

When all 4 TRANS part numbers have been processed, the program stops.

Assemble and run the program, follow the program by slow tracing, and you’ll see the process. Use ZT to find the results in the MASTER table (get the locations of the “on hand” and “number ordered” from the assembly listing).

Note: In this and other lessons you’ll see an end instruction that “jumps to itself.” Press BREAK to stop execution.

If you can follow the code in this program, you’re doing very well. We have covered a great deal of ground in the past lessons and this program incorporates most of the concepts we’ve discussed in the past lessons. A great deal of Z-80 assembly-language “code” is no more difficult than the program above.

To Sum It All Up

To recap what we’ve learned here:

- IX and IY can be used as “register indirect” pointers, but may also be used as “index registers”
- When used as index registers, IX and IY use an 8-bit displacement value

- The displacement value, when added to the contents of the index register, forms an “effective address” that points to the operand for the indexed instruction
- Data can be referenced up to 128 bytes back or 127 bytes forward from the index “base”
- Indexing can be used to advantage in table and other operations

For Further Study

Instructions using IX or IY indexing (Appendix V)



Lesson 15

Table Techniques Part III

Load LESS15 from disk.

In the previous lessons, we've covered most of the programming techniques that we can use to access data in tables; using the indexing addressing mode plays an important part.

The tables before this lesson were "scanned" one entry at a time to find data; we didn't know exactly where in the table we'd find the "search key," and had to methodically go through all the entries until we found the one we were looking for. These types of tables are called "unordered," because the entries don't have any logical order.

In this lesson we'll look at tables with an "order."

Types of Orders

In ordered tables, the order may be in alphabetic order, like a phone book, numerical order, historical order, or other scheme. Also, the order may be "ascending," like the phone book, or "descending." Descending would be a phone book printed from Z to A.

The most common order in assembly-language tables is alphabetic order, as in other types of programming. Usually the sequence is "ascending," from A to Z.

Actually, the "alphabetic" order really includes all ASCII characters, so it's really "alphanumeric" and special characters. Look at Appendix VII to see the ASCII codes for alphabetic, numeric, and special characters. The order we'll be using here will be based on these codes.

An example of data sorted on these codes is shown in Figure LESS15-1. Note that as you would expect, A through Z and 0 through 9 are kept in order, but that "upper case" comes before "lower case," that blanks come before anything else, and that special characters are somewhat scattered around.

A A R D V A R K	␣	B I L L	␣	␣	␣	
A A R D V A R K	,	A N T H O N Y				
A A R D V A R K	,	a n t h o n y				(lower case last)
A A R D V A R K	-	a n t h o n y				(- after comma)
A B L E		J A M E S	␣	␣	␣	␣
A B L E	!	J A M E S	␣	␣	␣	␣
A B L E	,	J A M E S	␣	␣	␣	␣
A B L E	,	C H A R L I E	␣	␣	␣	␣
A B L E	,	C h A R L I E	␣	␣	␣	␣
A B L E	,	J O H N	␣	␣	␣	␣
A B L E	,	J O H N	␣	C	␣	␣
A B L E	,	J O H N	␣	C A R T E R		(period before C)

␣ = BLANK

Figure LESS15-1. A Sorted List

Sorting

How do tables get in order? Suppose that we're entering a list of names as we think of them. How do we order them? This process is called "sorting" and it means that we're ordering the data according to some scheme, usually alphanumeric.

Because sorting long tables or lists of data involves a great deal of processing and sorting is such a

15 Table Techniques Part III

common technique, there are many different sort schemes — the bubble sort, the two-buffer sort, the Shell and Shell-Metzner sorts, and others. You might want to look at some of these other schemes in detail. You can find the “algorithms” in computing magazines.

One of the most common sorts used in assembly language is the “bubble sort.” The bubble sort operates as shown in Figure LESS15-2.

ORIGINAL	FIRST PASS			SECOND PASS		THIRD PASS	SORTED!
5	5	5	5	2	2	1	
16	2	2	2	5	1	2	
2	16	1	1	1	5	5	
1	1	16	8	8	8	8	
8	8	8	16	16	16	16	


 INDICATES SWAP
TOOK PLACE

Figure LESS15-2. Bubble Sort Operation

A table of entries is originally “unordered.”

The sort starts with the first two entries and compares them. If the second entry is of lower “order” than the first, the two entries are “swapped.” If the second entry is equal to or greater than the first, no swap is made.

The second and third entries are now compared, and a swap is made or the entries are left unchanged. This process continues until the last two entries in the table are compared.

At the end of the first pass, the table entries are usually not ordered. Other passes that repeat the compare and swap process have to be made. The swapping continues until the entries are in alphanumeric order.

As the “lighter” entries “bubble” to the top, this sort is called a bubble sort.

We’ve programmed a bubble sort below. Use the Lesson File, or enter the source yourself:

```

100 ; BUBBLE SORT
110 BUBSRT      LD      E,0           ;set pass # to 0
120 BUBO10      LD      IX,0B000H     ;point to table
130             LD      B,31          ;32 locations
140             LD      C,0           ;set change flag to 0
150 BUBO20      LD      A,(IX)         ;get first entry
160             CP      (IX+1)        ;test next
170             JR      Z,BUBO30      ;go if A=B
180             JR      C,BUBO30      ;go if A<B
190             LD      D,(IX+1)       ;get second entry
200             LD      (IX),D        ;swap B to A
210             LD      (IX+1),A      ;swap A to B
220             LD      C,1           ;set “change”
230 BUBO30      INC      IX           ;pnt to next pair
240             DJNZ    BUBO20        ;go if not one pass
250             INC      E            ;increment pass #
260             LD      A,C           ;get change flag
270             OR      A             ;test for change
280             JR      NZ,BUBO10     ;go if change occurred
290             END

```

Assemble the source code above until you get an error free listing.

The program works with the “experiment area” in B000H through B01FH. It assumes that this area is a table of data; each entry in the table is a one-byte entry, so there are a total of 32 bytes in the table.

The bubble sort sorts the 32 bytes, putting the “lower” order bytes at the beginning.

The experiment area has been filled with “unsorted” characters from the load of the Lesson file. To see how it works, set a fast speed by

ZS 9999

trace the B000H area in ASCII by

ZTT B000

and execute the program, keeping your eye on the trace area.

Notice how the data has been ordered in alphanumeric order. The bubble sort is inherently slow, but is fun to watch.

Try the program again at a lower speed by using a slow setting of ZS and filling the B000H area with any data you’d like. Execute the program again and keep one eye on the E register, which holds the pass number, one eye on IX, which points to the table address, and Whoops! . . . both on the trace area to see the data moving.

Try different data patterns. You may try the ZTT mode for ASCII if you’d like, but this will be most valuable when all the bytes are valid ASCII characters (otherwise you’ll get a period for the character).

Let’s look at the program.

There are really two loops here, an outer and inner.

The inner loop goes from BUB020 through the DJNZ BUB020 and is used to “scan” down through the table from beginning to end. At BUB020, IX points to the start of the table at B000H. The B register holds 31 and is used to count the number of comparisons. There are 31 pairs in the table 0,1/1,2/2,3/ . . . 29,30/30,31, one less than the size of the table. The C register is set to 0 at the beginning of each new inner loop pass. If **any** swap occurs, then it is set to 1. At the end of the pass, if C is still set to 0, no swap has occurred and the table is sorted.

The outer loop starts at BUB010 and goes to the end of the program. This loop initializes IX, B, and C for each new pass at the beginning of the pass. At the end of the pass, it increments the pass number in E and then tests the change flag in C. If a change occurred, the table is still not sorted, and another pass is made by looping back to BUB010.

About the only “tricky” part of the program that hasn’t been covered up to this point is the CP instruction action. We’ve talked about using CP, but haven’t really considered all the details in regard to the C flag.

Using the Carry Flag for Comparisons

The CP can be used to compare two bytes in **unsigned** fashion as follows: A compare compares A with another operand, call it B, by subtracting B from A: A-B.

If A is less than B, the Carry flag will be set (C). If A is equal to or greater than B, the Carry flag will be reset (NC). In this case, a JR was made if the C flag was set or A was less than B, meaning that no swap had to be made. **This condition always applies for a CP, or for a SUB, regardless of whether it is an 8-bit CP or SUB, or a 16-bit SBC.**

You can see how this works by using a few more of the features of the ALT program. Use the ZM (modify

memory) feature to change memory locations B000H and B001H. Location B000H will contain the “A” operand, and location B001H will contain the “B” operand. A typical example:

```

ZM B000=45 33      (change location B000H from 45 to 33)
BOO1=78 34        (change location B001H from 78 to 34)
BOO2=67 ENTER      (stop change)

```

If you are tracing the B000H area, you should see the memory locations change to the data you entered.

Now use the ZR feature to modify the IX register to point to location B000H:

```
ZR IX=B000
```

Now use the Breakpoint feature to “breakpoint” the JR C,BUB030 instruction:

```
ZB XXXX          (set the breakpoint at the location of the JR C,BUB030)
```

The Breakpoint sets up a point at which control will be returned back to the ALT command mode. The instruction breakpointed will not be executed. In this case, the LD A,(IX) at BUB020 and the CP (IX+1) will be executed, comparing the A operand with the B operand.

Now execute the two instructions by

```
ZX XXXX          (XXXX is the location of BUB020)
```

The breakpoint will be reached, and the results will be displayed on the screen. Look at the state of the Carry flag after the breakpoint is reached. Try various numbers in B000H and B001H to test the CP comparison. The carry should be set if A is less than B or reset if A is equal to or greater than B.

Now here’s a short quiz for you: How would you compare the A and B registers and

- Jump if A was greater than B?
- Jump if A was less than or equal to B?
- Jump if B was greater than A?

We’ll give you a second to think of the instructions....

Got them? Here are the answers:

To jump if A was greater than B, do something like:

```

                                CP          B          ;compare A to B
                                JR          C,NEXT      ;go if A<B
                                JR          Z,NEXT      ;go if A=B
                                JR          OUT         ;go if A>B
NEXT                            ...              ;A=B or A<B

```

In this case, if there was no carry, A was equal to or less than B, and we had to test the “equal to” condition first.

To jump if A was less than or equal to B:

```

                                CP          B          ;compare A to B
                                JR          C,OUT       ;go if A<B
                                JR          Z,OUT       ;go if A=B
NEXT                            ...              ;A>B here

```

To jump if B was greater than A: This is really the same as A is less than B, which is our original instruction:

```

                                CP          B          ;compare A to B
                                JR          C,OUT       ;go if A<B

```

A Bubble Sort of A Two-Byte Entry Table

One of the best ways to learn anything is by doing, so we'll continue in this vein with the following problem: Suppose you had a table made up of two-byte entries at locations B000H through B01FH, as shown in Figure LESS15-3.

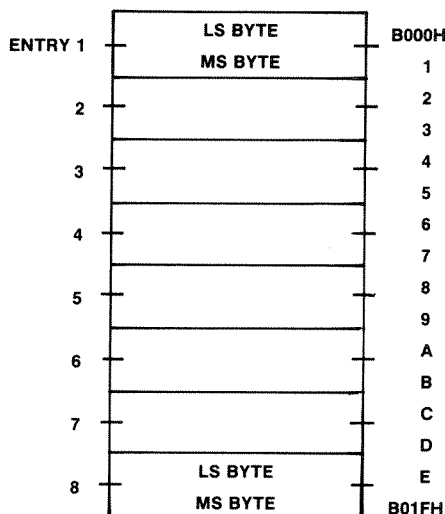


Figure LESS15-3. Two Buffer Sort Example

Each two-byte entry would be in standard Z-80 16-bit format of least significant byte followed by most significant byte.

How would you write a bubble sort to work with these two-byte entries?

Take some time now and try to come up with a program to do this, based on the earlier program. Hints: You'll want to use the HL register as the "accumulator" and load another register pair for the second operand. You could use either IX or IY to point to the table entries, incrementing the index register by two after each comparison. You'd also need to hold a "change" flag in another register and probably a loop counter in another that holds one less than the number of pairs in the table.

Another hint: The SBC HL,XX will set the carry flag in the same way as the CP.

Write the program, edit it, and assemble, and then run it with sample data. You can always reload if the program "blows up," but it shouldn't.

Here's the answer: There are many ways to do this, so if your program is different than this one, don't let it bother you. It may actually be better, in terms of speed or memory storage. The one we've come up with is:

15

Table Techniques Part III

300 ; BUBBLE SORT FOR TWO-BYTE TABLE

310	BUBSRT	LD	IX,B0000H	;point to table
320		LD	B,15	;15 pairs
330		LD	A,0	;set change flag to 0
340	BUB011	LD	L,(IX)	;get first entry
350		LD	H,(IX+1)	
360		LD	E,(IX+2)	;get second entry
370		LD	D,(IX+3)	
380		OR	A	;reset carry
390		SBC	HL,DE	;compare
400		JR	Z,BUB021	;go if A=B
410		JR	C,BUB021	;go if A<B
420		ADD	HL,DE	;restore HL
430		LD	(IX),E	;store B to A
440		LD	(IX+1),D	
450		LD	(IX+2),L	;store A to B
460		LD	(IX+3),H	
470		LD	A,1	;set "change"
480	BUB021	INC	IX	;pnt to next pair
490		INC	IX	
500		DJNZ	BUB011	;go if not one pass
510		OR	A	;test for change
520		JR	NZ,BUBSRT	;go if change occurred
530		END		;end

We'll leave it up to you to enter this program and execute it with data in the B000H area, after deleting lines 100 through 290. If you have trouble understanding how this program works, go back to the last lesson on indexing.

To Sum It All Up

To review what we've learned in this lesson:

- Tables may be ordered in alphanumeric or other order.
- The order may be "ascending" or "descending"
- Many tables are in alphanumeric and ascending order
- Various types of sorts are used to put tables in order
- The bubble sort works by comparing pairs of entries in a table and swapping if the first is of higher value than the second
- The Carry flag can be used to compare two 8-bit or two 16-bit values in CPs, SUBs, or SBCs
- The Carry flag is set (C) if the first is less than the second and reset (NC) if the first is equal to or greater than the second
- Breakpointed instructions will turn control back to the ALT command mode and can be used to selectively execute instructions

For Further Study

Carry flag setting for CP, SUB, SBC (Appendix V)

Lesson 16

Block Compares

Load LESS16 from disk.

In a previous lesson we looked at several “block move” instructions, the LDIR and LDDR. In this lesson we’ll look at some related instructions, the “block compare” instructions.

The block compares are CPIR and CPDR, along with two similar block compares, the CPI and CPD. Basically, all four are instructions that will rapidly scan a list of bytes, looking for a single given value, similar to some of the table techniques that we’ve been discussing in the previous lessons.

To get a fast idea of how the block compares work (but without the use of block compare instructions), use the existing code from the Lesson File:

```
100 ; BLOCK COMPARE
110 COMPAR    LD      HL,B000H    ;start of data
120          LD      BC,32        ;size of block
130 COM010    LD      A,23        ;search value
140          CP      (HL)        ;test for value
150          INC     HL          ;bump pointer
160          DEC     BC          ;decrement count
170          JR      Z,COM020     ;go if found
180          LD      A,B         ;test done
190          AND     C
200          CP      0FFH        ;0FFH if BC=FFFFH
210          JR      NZ,COM010    ;go if more in block
220          OR      A           ;set NZ
230 COM020    END                ;end
```

The Lesson File puts sample data into the B000H area. This program searches the 32 bytes from B000H through B01FH for the value in A.

Assemble and execute the program.

At the end of the program, the HL register should point to one more than the value at B014H. The Z flag should be set to Z (1). The BC register should contain 000BH.

By now you should be able to follow programs such as this with a little bit of head scratching. In the program above, HL points to the B000H area and is incremented by one each time through the loop. BC contains the number of bytes in the block to be searched. A contains the search “key,” the value to be found.

Each time through the loop, the contents of A are compared with the value from the B000H area. Before the jump that tests the results of the compare, HL is incremented to point to the next byte in the table and BC is decremented. If the two values match, the Z flag will be set and a jump is made to COM020.

If the two values do not match, the contents of BC are checked. If they contain FFFFH, the count has been decremented down past zero. The test is made by ANDing the B and C registers and comparing the result to 0FFH. Only when both equal 0FFH will the result be 0FFH. If BC=FFFFH, then the entire block has been searched and the key in A has not been found.

One way or another, the END statement is reached. If the end is reached with the Z flag set (Z), HL points to the found byte + 1. If the Z flag is reset (NZ), the key has not been found.

Run the program above several times using different values in the A register by changing the immediate load of LD A,23 and reassembling. Note the cases where the search key is found and the contents of the HL pointer.

The CPIR Instruction

The program above is almost an exact duplication of the way the CPIR instruction works. An equivalent program using CPIR is shown here. Delete lines 100 through 230 and enter this code:

```

240 ;BLOCK COMPARE USING CPIR
250 COMPA1    LD      HL,0B000H    ;start of data
260          LD      BC,32         ;size of block
270          LD      A,23          ;search value
280          CPIR                     ;test for value
290          END                    ;end

```

Assemble and try this program the same way that you ran the previous block compare. The settings of BC, HL, and the Z flag should be the same.

The CPIR saves quite a few instructions. It does the entire search of the list or table in one instruction!

Here's a reiteration of how to use CPIR:

- Set HL to point to the start of the block
- Set BC to the number of bytes in the block
- Set A to the search key
- Do CPIR
- If Z, then HL points to **one more** than the value in the block. If NZ, then the key was not found.

The CPDR Instruction

The CPIR instruction mnemonic stands for "Compare and Increment"; the CPDR is "Compare and Decrement." In the CPDR case, the block search is made from end to beginning instead of from beginning to end. To use the CPDR:

- Set HL to point to the last address of the block
- Set BC to the number of bytes in the block
- Set A to the search key
- Do CPDR
- If Z, then HL points to **one less** than the address of the value in the block. If NZ, then the key was not found.

Suppose we wanted to write a program to search the B000H area from end to beginning. What would it look like? Try your hand at it, before continuing.

Got it? See how it compares with the program below:

```

300 ;BLOCK COMPARE USING CPDR
310 COMPA2    LD      HL,0B01FH    ;end of data
320          LD      BC,32         ;size of block
330          LD      A,23          ;search value
340          CPDR                     ;test for value
350          END                    ;end

```

Enter this program after first deleting lines 240 through 290. Assembly and execute to see how the program works; use different data values in A (change the immediate load instruction) and different data in the B000H area.

As we described above, the HL register in the CPDR case points to one less than the address, if the value is found.

Using CPIR and CPDR to Scan A Table

At first it seems like having the HL register pointing to one greater (CPIR) or one less (CPDR) than the found value is a nuisance. However, this can be turned into an advantage. Suppose that we wanted to count the number of times a certain value was found in a table. We could do it fairly easily by using CPIR or CPDR.

Enter this code after deleting lines 300 through 350:

```

360 ; SCANNING A TABLE WITH CPIR
370 SCNTAB      LD      HL,OB000H      ;start of table
380             LD      BC,32          ;32 bytes
390             LD      A,1            ;search key
400             LD      D,-1           ;initialize count
410 SCN010      INC      D              ;bump count
415             CPIR                     ;search for key
420             JR      NZ,SCN020      ;go if not found
430             JP      PE,SCN010      ;go if not done
440             INC      D              ;last increment
450 SCN020      END                      ;end, D holds count

```

This program uses CPIR to scan the B000H through B01FH area. If the search key in A is not found, the first CPIR will immediately transfer to the END at SCN020. If at least one value is found that corresponds to the search key, then the count in the D register is incremented by one and the program loops back to SCN010 for the next try; at this point HL, BC, and A are properly set to look for the next occurrence of the value!

What is the JP PE,SCN010 instruction all about? That's part of the CPIR and CPDR operation that we left out in the above discussion. The P/V flag is set (1) as long as the byte count in BC is not zero. When the byte count reaches 0 (when the end of the table has been reached), the P/V flag is reset (0).

The JP PE,SCN010 conditional branch tests the state of the parity/overflow flag. **PE corresponds to the P/V flag being set, while PO corresponds to the P/V flag being reset.** Unfortunately, the mnemonics for the P/V flag are not as descriptive as the mnemonics for other conditions, and you'll have to do a little translation here.

It's possible, by the way, to have both a "match" (found key) and the end of the table. In this case, the search key value is in the last entry of the table.

Try running the program above with various search key values (change the immediate load of A) and different data. D will hold the number of times the search key value is found in the table.

CPI and CPD Operation

CPI is very similar to CPIR and CPD is very similar to CPDR. CPI searches forward, while CPD searches backwards. The registers are set up the same way before the CPI and CPD.

About the only difference between the two sets of instructions is that CPI and CPD execute only one "iteration" instead of going through the entire block! It's up to the programmer to loop back to complete the scan of the block. To implement a block search using CPI, for example, we'd have something like:

16 Block Compares

;BLOCK COMPARE USING CPI

COMPAR	LD	HL,0B000H	;start of data
	LD	BC,32	;size of block
	LD	A,23	;search value
COMO10	CPI		;test for value
	JR	Z,FND	;go if found
	JR	PE,COMO10	;go if more
NOTFND	...		;not found here
FND	...		;found here

The CPD would work the same way in reverse. Whenever PO occurs, the entire block has been scanned. The Z flag is checked before the P/V flag, allowing for the case of the value being in the last entry of the table.

What is the good of CPI and CPD? Why not simply use CPIR and CPDR instead of using the “overhead” of looping back to the CPI and CPD?

The chief reason is that the CPI and CPD allow the automatic incrementing of HL and decrementing of BC to be “broken out” from a single instruction. We can now do one compare at a time and add other instructions between the compare and looping back to the CPI or CPD.

Here’s a good example of why CPI and CPD are useful. Suppose that we have a table with 3-byte entries, as shown in Figure LESS16-1. The first byte of each entry is a one-character command abbreviation. The next two bytes are the “jump location” for the command. We can use the CPI to easily scan the table for a given command.

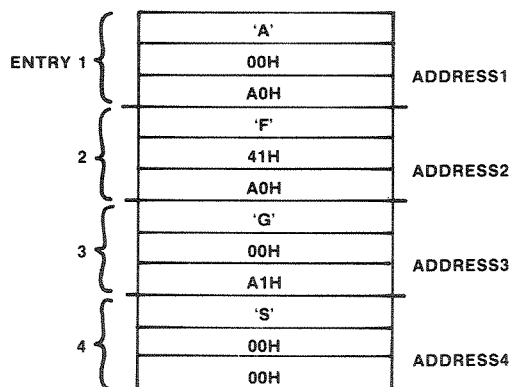


Figure LESS16-1. CPIR Use

```

460 ; SCAN TABLE USING CPI
470 SCNCPI      LD      HL, TABLE      ;start of table
480             LD      BC, 4           ;number of entries
490             LD      A, (0B000H)     ;search character
500 SCNO11      CPI                     ;compare entry
510             JR      Z, SCNO21       ;go if found
520             INC     HL              ;point to next
530             INC     HL
540             JP      PE, SCNO11      ;go if not done
550             LD      HL, 0           ;mark not found
560 SCNO21      DEC     HL              ;start of entry
570             LD      (0B001H), HL    ;store pointer
580             JR      STOP            ;end
590 TABLE      DEFM     'A'           ;ABORT command
600             DEFW     0A000H         ;ABORT program
610             DEFM     'F'           ;FIND command
620             DEFW     0A041H         ;FIND program
630             DEFM     'G'           ;GO command
640             DEFW     0A100H         ;GO program
650             DEFM     'S'           ;SMITHEREENS command
660             DEFW     0              ;reload system
670 STOP        END                     ;end

```

Enter the program above after first deleting source lines 360 through 450. Assemble the program.

Now enter a one-character command into location B000H. ASCII A, F, G, S are 41H, 46H, 47H, and 53H, respectively, or you may enter a character that is not present in the table.

Execute the program, and you'll find the location of the command "jump" table entry in B001H and B002H, in reverse address format. Try both valid commands and invalid commands. An invalid command will result in a -1 in locations B001H and B002H.

This program used the CPI to an advantage. After each compare in a CPI, the HL pointer was incremented by 2. (Don't forget that the CPI automatically incremented by one.) The BC register pair contains the **number of entries** rather than the number of bytes; when this is decremented down to 0 by the CPI, all entries in the table have been searched.

Couldn't we have used a CPIR in this program? Not really. The CPIR would have searched the table for a given key — all bytes of the table, both the one-letter command abbreviation and the address alike. If we were searching for an "A" (41H), then the CPIR would have found an erroneous "A" in the address byte for FIND; the first byte of the FIND program address is also a 41H.

To Sum It All Up

To recap what we've learned here:

- There are four block compares, CPIR, CPDR, CPI, and CPD
- The CPIR searches for a given search key
- The CPIR is a forward search of a block
- CPIR preparation requires that HL be set to the start of the block, BC to the size of the block, and A contain the search key
- After the CPIR, Z is set if the key is found, and HL points to the location of the value plus one

16

Block Compares

- The CPDR works in similar fashion to CPIR, but from the block end to the beginning; HL points to one less than the value if found
- CPIR and CPDR can be used to scan a table for more than one entry
- The P/V flag is set to “PO” after a CPIR or CPDR to mark the end of the block
- The CPI and CPD work the same as CPIR and CPDR except that only one iteration is made
- The CPI and CPD are useful in cases where a search is made of multiple-byte entries in tables

For Further Study

LDI, LDR

Lesson 17

Subroutine Use

Load LESS17 from disk.

Up to this point in the lessons, we've been using programs that use loops to execute code over and over again. There is another type of program structure that allows us to use code more than once — the subroutine.

Subroutine Basics

Subroutines may be from one to thousands of instructions long. If any set of instructions is executed more than one time, the instructions may be made into a subroutine in one place in memory to save memory space and to save time in "coding."

A subroutine is "called" by a CALL instruction; an RET instruction marks the end of the subroutine. A simple "timing loop" subroutine, for example, might be something like:

```

; DECREMENT HL DOWN TO -1 AS TIMING LOOP
TIMELP      LD          BC,-1          ;for decrement
TIMO10      ADD         HL,BC          ;count in HL-1
            JP          C,TIMO10      ;go if not thru 0
            RET              ;return

```

This timing loop might be called every time that you wanted a delay in the program; it delays about 1/1000th (1 millisecond) for every 100 counts in HL, to show you how fast assembly-language is. A typical call would be

```

LD          HL,200          ;load timing count
CALL       TIMELP          ;delay 2 milliseconds
...
            ;return here

```

As you can see from the above, a return is made to the instruction after the CALL. The Z-80 records the location of the next instruction after the call by taking the contents of the PC (Program Counter) and saving it in the "stack," an area of RAM. The RET instruction retrieves the location from the stack and puts it back into the PC to cause a jump back to the location after the CALL.

Let's investigate what the stack is and where it is located to see how the CALL and RETURN work. Enter the program below, or use the existing source code in the Lesson File:

```

100 START   LD          SP,B800H      ;load Stack Pointer
110         LD          HL,100        ;delay count
120         CALL       TIMELP        ;call delay
130 HERE    JR          HERE         ;loop
140 ; DECREMENT HL DOWN TO -1 AS TIMING LOOP
150 TIMELP  LD          BC,-1          ;for decrement
160 TIMO10  ADD         HL,BC          ;count in HL-1
170         JP          C,TIMO10      ;go if not thru 0
180         RET              ;return
190         END              ;end

```

Assemble and get an error-free assembly. Now trace the experiment area at B7F0H, and set the speed by

ZS 9999

Here's what should happen when you execute the program: The Stack Pointer register should be loaded with B800H; you should see this change in the register trace area.

17 Subroutine Use

Now the HL register is loaded with 100. Next, the CALL jumps to location TIMELP. The jump action is the same as a JP as far as the transfer of control.

The CALL, however, does one more thing. If you look at the locations B7FEH and B7FFH, you'll see the address of the return point after the CALL. The CALL has stored the return point by using the SP register as a "register indirect" pointer.

When you see the return address stored, look at the contents of the Stack Pointer. It started off with B800H, but after the CALL, it reads B7FEH.

The SP register is automatically decremented by 2 for every return address stored in the stack area. The "stack area" in this case started at B7FFH and "builds" down, as the CALL decrements the SP register by 2.

While the timing loop is delaying, keep your eye on the SP. As soon as the timing loop is over, the return address is loaded into the PC from the stack as the RET is executed. At the same time, the SP register is **incremented** by 2. After the RET, the SP again points to location B800H, and the instruction after the CALL is executed.

Execute the program and observe the actions. Press BREAK to get back to Command Mode.

A clarifying point about something that may be puzzling you: The timing loop subroutine indeed delays about 1/1000th of a second in normal Z-80 assembly-language programs. However, in the ALT we're "interpreting" each instruction, and this adds quite a bit more time to execution. The advantage, of course, is that we can see things happening and provide control over instructions that might do erroneous jumps and stores.

The Stack

The operation above is pretty typical of how the stack is used for subroutine calls. The stack can be any area of memory that will be unused by the system or by your own program. Of course, it has to be RAM, as we're both reading and writing into it by RETs and CALLs.

The stack area is normally set once at the beginning of the program, by the

```
LD      SP,0B800H
```

or similar instruction. Note that the SP is loaded with **one more** than the first stack location used. The SP is always **decremented first**, before the store of the CALL address.

Normally, you don't have to worry too much about setting the SP. BASIC, TRSDOS, and other programs **always** set the SP to the stack area used in their programs, and it's available for everybody else's use too. Sometimes, though, you want to control the stack yourself, and in that case you'll use the LD,SP to point to your own stack area.

Normally the stack area should be about 100 bytes or so. All this means is that you must make certain that the SP is set to the last location of a memory area that won't be used by anything else.

Nested Stack CALLs

Not only can you CALL a single subroutine, but that subroutine can CALL another subroutine, and that subroutine can CALL another, and so forth. How many CALLs can you make? Theoretically, as many as you want. Each time you make a CALL, another return address is saved in the stack, and that takes 2 bytes. If you have 10 "levels" of subroutines, you've used up 20 bytes of the stack, and the SP register points to the original location minus 20 bytes.

Let's see an example of a "nested" set of subroutines. Delete the previous source lines and enter the program below:


```

200 ; NESTED SUBROUTINES
210 NESTSR      LD          SP,B800H      ;stack area
220             CALL        SUBR1        ;call subroutine 1
230 HERE1       JR          HERE1        ;return point 1
240 SUBR1       CALL        SUBR2        ;call subroutine 2
250             RET                      ;return point 2
260 SUBR2       CALL        SUBR3        ;call subr 3
270             RET                      ;return point 3
280 SUBR3       CALL        SUBR4        ;call subr 4
290             RET                      ;return point 4
300 SUBR4       RET                  ;return
310             END                    ;end

```

Assemble this program, and then execute at a very slow speed, while tracing the B800H area (ZT B7F0). Keep your eye on the B800H area from B7FFH on down and also observe the SP register.

What did you see?

You should have first seen the SP load with B800H. It points to one more than the first stack area location. The CALL SUBR1 should have stored the address of the JR HERE instruction into locations B7FEH and B7FFH, in reverse address format.

SUBR1 consists of another CALL, to SUBR2. This should have stored the address of return point 2 into locations B7FCH and B7FDH.

SUBR2 consists of another CALL, to SUBR3. This should have stored the address of return point 3 into locations B7FAH and B7FBH.

SUBR3 consists of a fourth call, to SUBR4. This should have stored the address of return point 4 into locations B7F8H and B7F9H.

At this point the stack area appears as shown in Figure LESS17-1. The four return points are "4 levels deep" in the stack, and the SP points to location B7F8H.

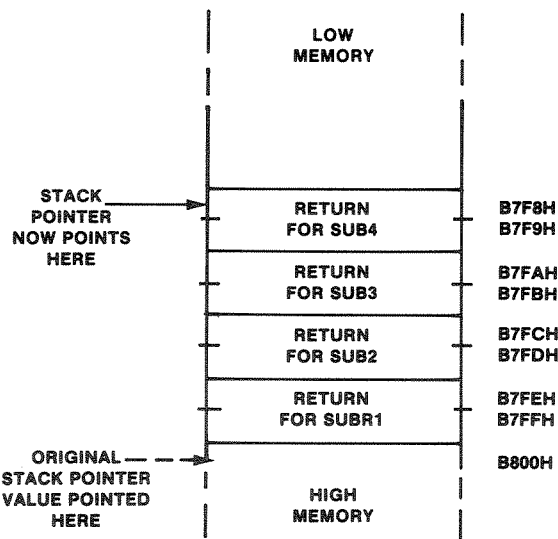


Figure LESS17-1. Stack Use on CALLs and RETs

When the RET in SUBR4 is executed, the RET causes return point 4 to be put into the PC, causing a JP to return point 4. The stack pointer now points to location B7FAH.

17 Subroutine Use

Return point 4 is also an RET, causing a JP to return point 3 and resetting the SP to B7FCH.

Return point 3 is also an RET, causing a JP to return point 2 and resetting the SP to B7FEH.

Return point 2 is also an RET, causing a JP to return point 1 and resetting the SP to B800H, the original setting.

If you didn't see all the actions of this program, execute it again, and watch the stack actions closely. There's nothing too mysterious about it.

This type of nesting is very common in many programs. Of course, this program does nothing except to illustrate the stack, and normal programs would have a great deal of "code" between the entry to the subroutine and the next CALL or RET.

Usually you won't use more than about 3 or 4 levels of subroutines. It's just too hard to keep track of where you are if you use more and usually not necessary.

A CALL for Every RET

If you look at the program above, you'll see a RET instruction for every CALL. If there wasn't an RET for every CALL, what would happen?

The answer is that the stack would get "out of sync." The wrong return point would be picked up on a RET. If you repetitively call the stack with this condition, you'll soon run out of stack and into another memory area, probably part of your program or a system program. That will put "garbage" (in programming terms) into the program area and cause your program or the system program to blow up.

This condition is called stack overflow (or underflow, depending upon which direction the stacks goes out of bounds) and is a common programming bug!

For example, if you had:

EXAMP	CALL	SUBR1	;call subroutine
	...		
SUBR1	...		;subroutine code here
	...		
	...		
	JP	EXAMP	;return s/b RET!

the JP back to EXAMP would cause SUBR1 to be called again, putting the return point in the next two bytes, and this would continue indefinitely with the return point being stored in lower and lower locations until the stack overflowed into a program area.

Types of CALLs and RETurns

We used a simple CALL and RET in the examples above. These were "unconditional" CALLs and RETs similar to the unconditional JPs.

There are a number of conditional CALLs and RETs, however, that can CALL a subroutine or return from a subroutine based on the results of a previous operation. These are very similar to conditional JPs.

The conditional CALLs are

CALL NZ	CALL PO
CALL Z	CALL PE
CALL NC	CALL P
CALL C	CALL M

The conditions have the same meaning as the conditions for the JPs and are easy to remember because of their similarity. As an example of a conditional CALL, you might want to call a subroutine if two numbers were identical:

CP	(IY+2)	;test A with B
CALL	Z,SUBR1	;go to subr if A=B

The conditional RETurns use the same conditions as the CALLs

RET NZ	RET PO
RET Z	RET PE
RET NC	RET P
RET C	RET M

Here again, the return can be made if A=B, if there is a carry, or another condition, depending upon your requirements.

In the next lesson we'll continue our discussion of the stack and show you some further stack actions. We'll also show you some usable subroutines that consist of more than CALLs and RETs!

Important Note

Use the ZRSP= command to reset the Stack pointer to -1 (FFFFH). If you do not do this, executing subsequent lessons may produce a 'JP, REF, OR STK OUT OF OBJECT' error. The Stack Pointer must be reset to high memory and doing a

ZRSP=FFFF

will properly reset the stack area.

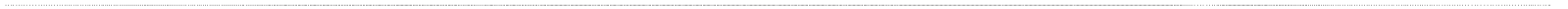
To Sum It All Up

To review what we've learned in this lesson:

- Subroutines are a collection of instructions that are used more than once and are in one location in memory
- Subroutines save memory and coding time
- Subroutines may be from 1 to many instructions long
- Subroutines are called by a CALL instruction and ended by an RET instruction
- A CALL saves the address of the instruction after the CALL in the stack
- AN RET gets the last return address from the stack and causes a jump back to that point
- The stack area in memory is any convenient memory area that can be set aside for stack actions
- The SP (stack pointer) points to the stack area
- The stack area "builds down"; each CALL stores 2 bytes of the return address into the next 2 lower stack locations
- Subroutines can be "nested" as often as required
- There must be an RET for every CALL
- There is one unconditional CALL and one unconditional RET; there are also conditional CALLs and RETs that use the same condition "codes" as the JP

For Further Study

Instruction formats for CALLs and RETs (Appendix V)



Lesson 18

Other Stack Operations

Load LESS18 from disk.

Important Note

Before starting this lesson, check to see that the Stack Pointer is properly set. It should read FFFF in the register line. If it does not, execute

ZRSP=FFFF

to reset it to FFFF. The SP will automatically change to another value after you execute a ZX command. This is normal.

In the last lesson we looked at CALLs and RETurns for subroutines. In this lesson we'll look at another use of the stack — storage of temporary results.

Stack Uses

In the last lesson we said that the stack was used for storage of addresses when a CALL was executed to a subroutine. There are actually three uses for the stack:

- Saving return address for CALLs
- Saving temporary data for PUSH and POP use
- Saving the interrupt point for interrupts

We've already described what happens in CALLs, but we'll talk about the other two uses here.

Interrupts

One of the two uses of the stack is somewhat esoteric. Interrupts are external or internal inputs to the system that signify a “real-world” event. One example might be an interrupt from a remote keyboard. During the time that no key is pressed, the program would run normally. When a key was pressed, however, an “external” interrupt could be generated that would cause the Z-80 cpu to stop processing after the current instruction and to jump to a special interrupt processing routine.

The interrupt processing routine is simply another assembly-language program that would take the required action for the external interrupt. In this case, the remote keyboard interrupt handler would probably read in the character from the keyboard and store it in a “buffer,” or storage area.

The interrupt action itself causes the interrupt point to be put into the stack in a very similar fashion to the CALL. An RETI, or Return From Interrupt, instruction at the end of the interrupt processing program would retrieve the interrupt point address from the stack and put it into the PC register in an almost identical action to a normal RET.

Interrupts are used primarily to let the Z-80 cpu number crunch away on a “background” job, such as running a business package, while a high-priority “foreground” task infrequently interrupts for a response to its action.

Another example of interrupts is the “real-time” clock interrupt. This interrupt occurs every 33 milliseconds or so (33/1000ths of a second) and causes the RTC interrupt processing routine to be entered. The RTC interrupt processing increments a count which is used to keep time in the system.

We won't be discussing interrupts any further in this text. Suffice it to say that they are used infrequently and in special systems applications.

PUSHes and POPs

The remaining use for the stack, however, is used all the time. A PUSH “pushes” a register pair onto the stack, while a POP “pulls” two bytes of data from the stack and puts it into a register pair.

18 Other Stack Operations

Why the “PUSH” and “POP”? The stack can be thought of as a “push-down stack” similar to a dinner plate stacker in Joe’s Greasy Spoon. A PUSH or CALL pushes a plate (two bytes) onto the stack. Further plates can be pushed on top of the previously pushed plates.

A POP or RET “pops up” the last plate (two bytes) pushed. Successive POPs or RETs pop up the stacker until no plates are left.

The available PUSHes are:

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
```

There is a corresponding POP for every PUSH — POP AF, and so forth.

Let’s see how the PUSHes and POPs work. Use the Lesson File for this program:

```
100;.....
110; SCAN TABLE FOR SMALLEST ENTRY SUBROUTINE      *
120;  ENTRY: (IX)=>TABLE                             *
130;          (B)=SIZE OF TABLE 1-255,0=256         *
140;  EXIT:  (IX)=>SMALLEST ENTRY IN TABLE          *
150;          (A)=SMALLEST ENTRY                     *
160;.....
170 SCANTY      LD          C,(IX)                   ;get first byte
180            PUSH        IX                       ;initialize stack
190 SCNO10      LD          A,(IX)                   ;get byte
200            CP          C                         ;test with lowest
210            JR          NC,SCNO20                 ;go if C>=A
220            LD          C,A                       ;new lowest
230            POP         DE                       ;previous pntr
240            PUSH        IX                       ;save this loc'n
250 SCNO20      INC         IX                       ;point to next
260            DJNZ        SCNO10                   ;go if not end
270            POP         IX                       ;get pointer
280            LD          A,C                       ;lowest in A
290            RET                                ;return
```

Don’t try to execute this program yet.

This program is a complete subroutine, as we talked about in the last chapter. It is a complete set of code to perform one specific function, to scan a table for the smallest byte in the table. If we had a table consisting of the one byte entries of 45,3,47,89,100,2,3,4,56, the subroutine would find the 2 entry in the table.

It’s common to divide a large programming job into many different subroutines, each performing a simple function. Another trick that’s used is to make each subroutine as “generic” in nature as possible. In this case, we didn’t limit ourselves to **any** table or **any size** table. We left the location and size of the table variable!

The location and size of the table are called **parameters** that are **input** to the subroutine. On entry into the subroutine, the index register IX points to the table start, and the B register holds the size of the table.

On **exit**, the IX register points to the location of the smallest entry in the table and the A register holds the actual entry itself. These are the **output parameters**.

"Parameters" might be called arguments, or "gozintas" and "gozoutas."

Because we've made SCANTY general, we can use it to scan any table of any size and it becomes a "general-purpose" subroutine.

SCANTY uses PUSHes and POPs for temporary storage. Let's see how it does this.

First of all, IX contains a pointer to the table on entry, and B contains the size of the table. The table is made up of from 1 to 256 one-byte entries. If B is initially 0, it denotes a 256-byte table.

The first thing that SCANTY does is to pick up the first table byte. This **may** turn out to be the lowest-valued byte, but probably isn't. The location of this byte (location in IX) is then **PUSHed** onto the stack. The stack now looks like Figure LESS18-1A.

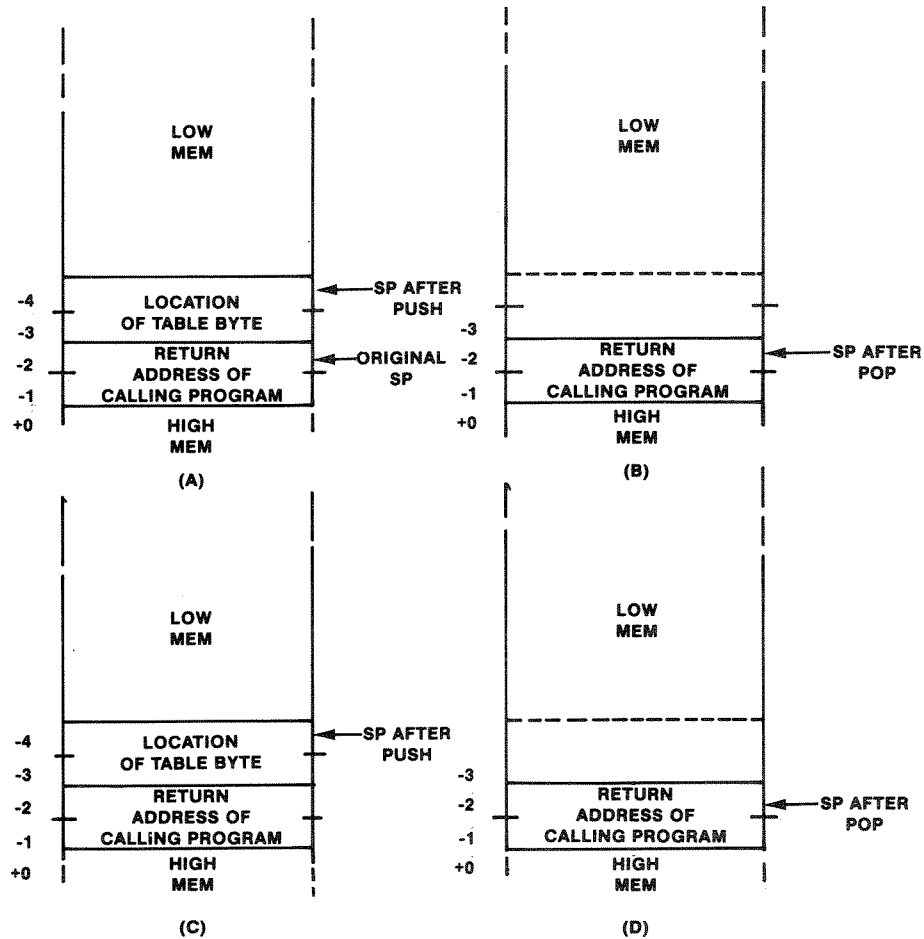


Figure LESS18-1. Stack Use on PUSHes and POPs

The loop from SCN010 is the main (the only) loop of SCANTY. It will go through the table and compare each byte with the current low byte. If any byte is lower than the current low byte, it will be stored in C and its location will be saved in the stack. We're starting off with the first byte and its location as an arbitrary initial value.

In the loop: The next byte is picked up and put into A. A is then compared with C, which holds the current lowest byte.

If the next byte is greater than or equal to C, SCN020 bumps the IX pointer to the next location and the count in B is decremented, causing a loop back to SCN010.

If the next byte is less than the current, it replaces the contents of C. Then, the POP DE pops the stack

18 Other Stack Operations

and puts the current location into DE. (See Figure LESS18-1B.) This is done only to reset the stack, to get rid of the current location. Next, the PUSH IX pushes the current location into the stack as shown in Figure LESS18-1C.

After the loop is over (the count in B goes to 0), the C register holds the lowest value in the table and the stack holds the location of that lowest value.

Note that when we say, “the stack holds . . .,” what we really are saying is that “the stack currently holds a 16-bit data value; no corresponding POP has been issued for it.” The stack might hold three or four different 16-bit values at this point, depending upon the program. Just remember that when the stack is used for temporary storage in this way, we’ll have to eventually use the values, or at least reset the stack pointer by dummy POPs.

Another interesting point: Although we used a PUSH IX to save the location in the stack, there’s no reason at all that we have to POP that value back into the IX register. In fact we’re POPping it into another register, the DE register pair. Once 16-bit values are in the stack, they can be POPped by AF, BC, DE, or HL at will.

After the loop, we POP the location into the IX register, as shown in Figure LESS18-1D. The lowest value in C is then put into A.

The last instruction is a RETURN that POPs the return address of the **calling** program. The term “calling program” means that another program has called the SCANTY subroutine.

Want to see how SCANTY works? Why don’t you write some code that will CALL scanty? Make the table start at location B000H and make it a size of 32. You can do it in 4 instructions, counting a 90 STOP JR END. Put the four instructions at source lines 60, 70, 80, and 90, before the SCANTY subroutine.

Got it? It should look like this:

60	LD	IX,B000H	;location of table
70	LD	B,32	;table size
80	CALL	SCANTY	;call scanty
90	STOP	JR	STOP

Assemble all of the source lines, fill the B000H area (from B000H through B01FH) with any data, and execute from line 60.

At the end of the execution the IX register should point to the lowest value in the B000H area, and the A register should contain the lowest value itself.

Multiple Subroutines

We now have a standard general-purpose subroutine for searching any table of 256 bytes or less for the lowest value. What can we do with it?

One idea that comes to mind is to use it for a two-buffer sort. Remember in a previous lesson when we implemented a bubble sort? We mentioned the two-buffer sort and said that it required twice the storage space because we needed an extra buffer. What the heck, memory is cheap, so let’s implement this version of a sort.

To do this, we’ll need two more subroutines. List lines 310 through 380 to see these:


```

300 ; STORE ENTRY IN A INTO (IY) LOCATION
310 STORE      LD      (IY),A      ;store entry
320           INC      IY          ;bump IY
330           RET              ;return
340 ;MARK OLD ENTRY WITH -1
350 MARK      LD      A,0FFH      ;-1
360           LD      (IX),A      ;store -1
370           RET              ;return

```

The first of these, STORE, stores the contents of A into the location pointed to by IY.

The second, MARK, marks the location pointed to by IX with a -1 (i.e. stores a -1 value in the location).

Given these three subroutines, can you construct a 10-instruction program to sort 16 bytes of data in locations B000H through B00FH into a second buffer at B010H through B01FH? Watch the B register! Try it before looking at the following code:

```

-----
380 ; TWO-BUFFER SORT
390 SORT      LD      IY,B010H    ;point to second buffer
400           LD      B,16        ;count
410 SORO10    PUSH    BC          ;save count
420           LD      IX,B000H    ;point to first buffer
430           LD      B,16        ;size of first buffer
440           CALL    SCANTY      ;find lowest entry
450           CALL    STORE      ;store in 2nd buffer
460           CALL    MARK       ;delete 1st buf entry
470           POP     BC          ;get count
480           DJNZ    SORO10      ;loop if not done
490           END                ;end

```

This sequence uses the three subroutines to do the sort. The first, SCANTY, finds the location and value of the lowest entry in the B000H buffer.

The second, STORE, stores the value in A into the B010H buffer by using IY as a pointer.

The third, MARK, sets the original location of the current value in the B000H buffer to -1. A value of -1 is used as a flag to say "this location already was a lower value and is not to be used from this point."

Before the CALL to SCANTY, IX and B must be initialized to point to the B000H buffer and for a count of 16, respectively.

About the only "tricky" part of SORT is using a PUSH BC to save the count before the CALL to SCANTY is made. If this were not done, the loop count in B would be destroyed by the table size parameter. After the three subroutines, the count in B is restored by a POP BC. This type of operation is very common in saving registers. Note that even though we had the count only in B, both B and C had to be pushed. There is no "PUSH B."

Run SORT at a slow speed after entering data in the B000H area. Execute from the starting address of SORT (ZX mmmm). You should see the B010H area fill up with ordered data and the B000H area fill up with 0FFH values. By the way, this SORT also works with 0FFH values (-1). Do you see why?

We'll be using PUSHes and POPs frequently from this point on, so review the Lesson from the start if you are confused about the PUSH, POP action.

To Sum It All Up

To review what we've learned here:

- Interrupts use the stack by pushing the return address for the interrupt in the stack
- PUSHes and POPs are used to temporarily store data in the stack
- Register pairs AF, BC, DE, and HL can be PUSHed or POPped
- There must be a POP for every PUSH, or the stack must be reset by “dummy” POPs
- Subroutines are often small segments of “generalized” code that work for many different sets of conditions
- Parameters are often passed to and from subroutines; these parameters are “arguments” that define the conditions for the subroutine
- Once data is in the stack, it is not related to any register pair; the same register pair or a different register pair may be used to retrieve the data

For Further Study

RETI, RETN action (Appendix V)

PUSH, POP formats — private study

Lesson 19

Shifting Data

Load LESS19 from disk.

Up to this lesson we've generally been working with one byte of data or two bytes of data. A lot of assembly-language code is concerned with manipulating **bits**. There are a number of instructions that allow us to set, reset, and test bits, and we'll cover them in following lessons. In this lesson, however, we're going to look at **shift** instructions, instructions that also manipulate bits, but do so a byte at a time.

Rotates

Rotates on A

The original 8080 predecessor of the Z-80 had four shift instructions — RLCA, RLA, RRCA, and RRA. These instructions were a class of shifts called **rotates**.

The four rotates operate on the A register, as this is the main accumulator, or working register, in the 8080 and Z-80.

The instructions rotate the A register either to the right or left, one bit at a time, as shown in Figure LESS19-1.

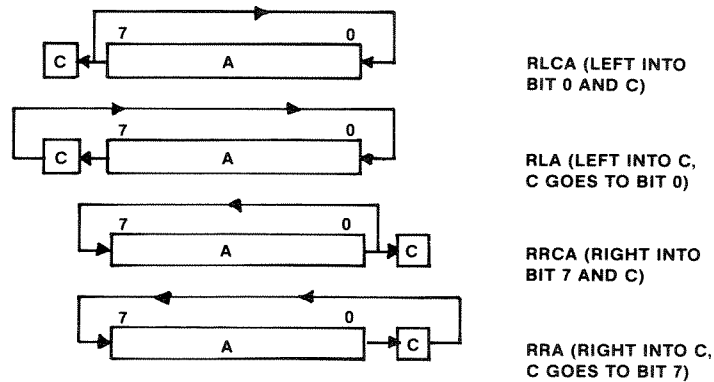


Figure LESS19-1. Rotates of A Register

Two of the instructions, RLCA and RRCA, rotate just through the A register itself. This is an 8-bit shift.

The remaining two instructions, RLA and RRA, work with the A register and Carry flag. This is a 9-bit shift.

To see how these work, enter the following program or use the Lesson File:

```
100 ;RLCA, RLA, RRCA, RRA OPERATION
110 ROTDEM    LD      A,OBH      ;load test pattern
120          LD      B,8         ;loop count
130 ROTO10    RLCA              ;rotate
140          DJNZ     ROTO10      ;do 8 times
150          LD      B,8         ;loop count
160 ROTO20    RRCA              ;rotate
170          DJNZ     ROTO20      ;do 8 times
180          LD      B,9         ;loop count
190 ROTO30    RLA               ;through carry
200          DJNZ     ROTO30      ;do 9 times
210          LD      B,9         ;loop count
220 ROTO40    RRA               ;through carry
230          DJNZ     ROTO40      ;do 9 times
240          END                ;end
```

19 Shifting Data

Set the speed to a slow setting and run the program. The following things should happen:

The RLCA loop will rotate the A register to the left, one bit at a time. As each bit is shifted around from bit 7 back into A through bit 0, you should also see the bit going into the Carry Flag. Eight shifts are done and at the end of the time, the A register should have the original value of 0BH or 00001011.

The RRCA loop will rotate the A register to the right, one bit at a time. As each bit is shifted around from bit 0 back into A through bit 7, you'll also see the bit going into the Carry flag. Eight shifts are done. At the end of the shifts, A will have the original value.

The RLA loop starts with the 0BH value in A. Nine shifts are done to the left, with bit 7 going into the Carry flag and with the previous contents of the Carry flag going back around into bit 0 of the A register. At the end of the shifting, A will contain 0BH.

The RRA loop starts with the 0BH value in A. Nine shifts are done to the right, with bit 0 going into the Carry flag and with the previous contents of the Carry flag going back around into bit 7 of the A register. At the end of the shifting, A will contain 0BH.

The RLCA, RLA, RRCA, and RRA instructions are used primarily to "align" data or sometimes to test the state of each bit in the A register. The only Flag that is set by these four instructions is the Carry flag, set according to the state of the bit shifted out of the register (the N and H Flags are reset).

As an example of this type of shift, suppose that we wanted to test the "parity" of the bits in the A register as a check on the validity of data read in from an external device. Suppose that we were reading ASCII bytes from an RS-232-C interface in the Model I or III. ASCII data is a 7-bit code, with the 8th bit (bit 7) either unused and set to 0 or 1 or used as a parity bit (see Appendix VI).

If even parity is used, then bit 7 is set to make the total number of 1 bits even. If odd parity is used, then bit 7 is set to make the total number of 1 bits odd. Sample ASCII data with the "parity bit" set is shown in Figure LESS19-2.

ASCII CHARACTER	7-BIT ASCII CODE	PARITY BIT ADDED	#1 BITS
A	1000001	0 1000001	2
B	1000010	0 1000010	2
C	1000011	1 1000011	4
D	1000100	0 1000100	2
M	1001101	0 1001101	4
Z	1011010	0 1011010	4

PARITY BIT
ADDED TO
MAKE TOTAL
NUMBER OF
1 BITS EVEN

Figure LESS19-2. ASCII Data with Parity

How could we count the number of 1 bits in the A register as a check on the received data byte?

A program for this is shown below. It is on the next portion of the Lesson File.

```

250 ; GET EVEN PARITY OF BYTE IN A
260 PARITY      LD          C,0          ;set parity to 0
270             LD          B,8          ;set loop count
280 PARO10      RLCA                 ;shift out bit
290             JR          NC,PARO20    ;go if a zero bit
300             INC          C           ;count # 1 bits
310 PARO20      DJNZ         PARO10      ;loop for 8 bits
320             LD          A,C         ;get count
330             AND          1          ;test parity
340             END                   ;Z if parity ok

```

This program will count the number of 1 bits in A by shifting out the bits 1 at a time, testing the Carry flag, and incrementing a count in C if the bit is a 1. At the end of the counting, C holds the number of 1 bits. An AND 1 then gets the least significant bit in C. This bit is 1 if the number of 1 bits is odd or 0 if the number of 1 bits is even, and is therefore the parity of the value in A. The Z flag is set according to the result. (It will be opposite to A.)

If the ASCII data is coming in with even parity, then an NZ condition after any test will indicate that the ASCII character was somehow garbled. If an ASCII A was changed from 01000001 (with the parity bit set to 0) to 01100001, then the parity would be off, as it would be odd instead of even (Z).

Try running the above program with various values in A. Make a manual count of the number of 1 bits. Store the value in A before execution by using the ZR A= command.

Other Rotates

As we mentioned before, RLCA, RLA, RRCA, and RRA were the “original” 8080 rotates. There are four **more** rotates in the Z-80, however, that work exactly the same as the four previous rotates except for the following:

- They can work with any register, A, B, C, D, E, H, or L or with a memory byte
- They set the flags somewhat differently
- They can shift a memory byte by using the (HL), or indexed addressing modes

The four rotates and their A register equals are:

Any Register Rotates	A Register Rotate
RLC	RLCA
RL	RLA
RRC	RRCA
RR	RRA

You can see that the new rotates have almost the same mnemonics as the A register rotates except that the “A” on the end is truncated (loped off). They work the same in the shift action, either doing an 8-bit shift around to the opposite end of the register or through the Carry Flag first.

You can see how it would be useful to have the capability to shift any register or a memory location. These four instructions must have an operand. To rotate the B register left one bit you’d have

```
RLC      B          ;rotate B
```

To rotate the memory location pointed to by (HL) right one bit through the Carry, you’d have

```
RR      (HL)        ;rotate memory
```

The C, Z, P/V, and S flags are set for these “any register” rotates. The Carry flag is set to the bit shifted out of the register or memory location. The Z flag is set if the shifting results in a zero. The P/V flag is set

19 Shifting Data

to the parity (number of 1 bits) after the shift (P/V=1 if even, P/V=0 if odd), and the S flag is set to 1 (M) or 0 (P) depending upon the bit in the sign bit after the shift.

Here's a student exercise for you: How would you set the Flags to indicate the zero/no zero state, the parity, and the sign of the contents of memory location B000H? You should be able to do this in several instructions. Try writing this short piece of code before continuing.

Got it?

One way to do it is by the following (delete lines 250 through 340):

```
350 ;TEST LOCATION B000H
360 TEST      LD      HL,B000H      ;point to 0B000H
370          RLC      (HL)          ;shift to left
380          RRC      (HL)          ;shift to right
390          END                    ;end
```

The RLC rotates out the contents of B000H left and then right again. Nothing changes except the flags. The C flag is set to the value in bit 7. The Z flag is set if the value in B000H is zero. The S flag is set if the original value in bit 7 was a one. The P/V flag is set to the parity of the byte in B000H.

Try various values in B000H and look at the results.

RLD and RRD

There are two other rotates that we should mention, as you've probably seen them in the instruction set and are wondering what they are.

The RLD and RRD rotate, but they rotate four bits at a time, as shown in Figure LESS19-3. The four bits in the least significant portion of the A register go into either the upper or lower four bits of the memory location pointed to by the HL register, used as an indirect pointer.

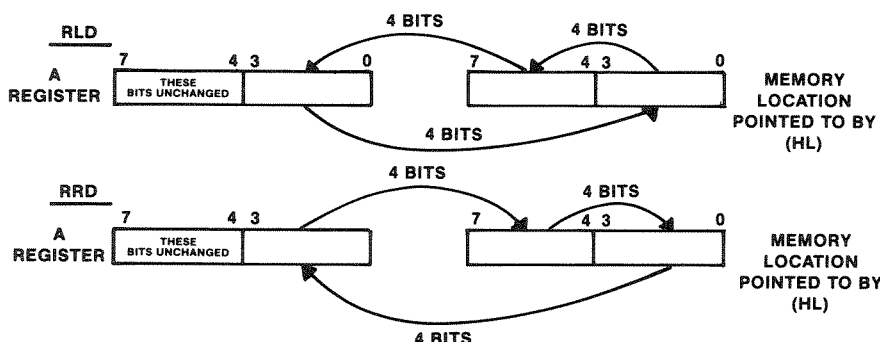


Figure LESS19-3. RLD and RRD Action

What is the purpose of such a strange shift? The RLD and RRD are used when working with “bcd,” or binary-coded-decimal data. Binary-coded-decimal data represents decimal digits of 0 through 9 in groups of 4 bits:

0000=decimal 0	0101=decimal 5
0001=decimal 1	0110=decimal 6
0010=decimal 2	0111=decimal 7
0011=decimal 3	1000=decimal 8
0100=decimal 4	1001=decimal 9

The binary-coded-decimal values of 1010 through 1111 are not allowed. In this method of representation, each binary byte holds 2 bcd digits. The four binary bytes

00010010 00110100 01010110 01111001.

for example, would represent the 8 bcd digits 12345679.

By using a special instruction, the DAA, we can add and subtract bcd digits and have them come out properly. We'll see how in another chapter.

The RLD and RRD, then, are geared toward moving bcd digits around, either right or left, and operate four binary bits at a time, but one bcd digit at a time.

Although at first glance it seems like this might be a very powerful instruction that could be used to shift in 4-bit chunks, it is not very flexible as far as register use or addressing.

To give you some practice in shifting, try the following problem: Write a program to **rotate** the memory locations from B000H through B00FH four bits to the right. The four bits on the left (location B000H) will rotate into the upper 4 bits of memory location B00FH, as shown in Figure LESS19-4. Do this with a "normal" 1-bit shift first, and if you feel ambitious, try it with an RRD. (Delete lines 350-390.)

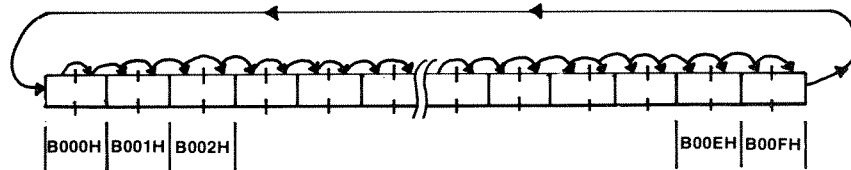


Figure LESS19-4. Rotate Example

```

400 ; SHIFT ONE HEX CHARACTER RIGHT B000H-B01FH
410 SHFHEX      LD      B,4          ;4 shifts
420 SHTO10      OR      A            ;reset carry
430             PUSH    BC           ;save count
440             LD      IX,B000H     ;start of data
450             LD      B,16         ;inner loop count
460 SHTO20      RR      (IX)         ;shift one bit to C
470             INC     IX           ;point to next byte
480             DJNZ    SHTO20       ;do 16 times
490             JR      NC,SHTO30    ;go if 0
500             LD      A,(IX-16)     ;get byte
510             OR      80H          ;set ms bit
520             LD      (IX-16),A     ;store
530 SHTO30      POP     BC           ;get count
540             DJNZ    SHTO10       ;go if not done
550             END                  ;end

```

The above program is one way to accomplish the shift. Try yours, if you did one — it may be shorter! The only trick here is that each RR (IX) shifts one bit to the right. The Carry from the previous shift goes into bit 7 on the shift, and a new Carry from bit 0 replaces the previous Carry. This new carry will go into bit 7 of the next memory location.

The inner loop shifts 16 locations one bit. The outer loop does 4 iterations of one bit.

Try the program (or yours). It should shift the data in B000H through B00FH one hex digit position to the right (use the ZT display mode).

19

Shifting Data

To Sum It All Up

- Rotates rotate either the A register (RLCA, RLA, RRCA, RRA) or any register or memory location (RLC, RL, RRC, RR) one bit to the right or left
- Rotates recirculate the bits back into the register or memory location from the other end
- Rotates may operate through the Carry flag (9-bit rotate) or without going through the Carry
- The Carry flag is always set to the state of the bit shifted out on a rotate
- The “any register” rotates also set the Z, S, and P/V flags
- BCD numbers are made up of binary-coded-decimal values in groups of 4 bits
- A BCD digit represents decimal 0 through 9 by values of 0000 through 1001
- BCD digits of 1010 through 1111 are invalid
- RRD and RLD are “BCD-shifts” that move 4 bits at a time

For Further Study

RLD, RRD (Appendix V)

DAA (Appendix V)

Lesson 20

More Shifting and Multiplication

Load LESS20 from disk.

There are several other types of shifts in the Z-80 that are used frequently. One of these is the “logical” shift, and the second is the “arithmetic” shift.

Logical Shifts

Logical shifts are different from rotates because they do not recirculate the data in the register or memory location. Logical shifts shift in zeroes in place of the data from the other end of the register or memory location as shown in Figure LESS20-1.

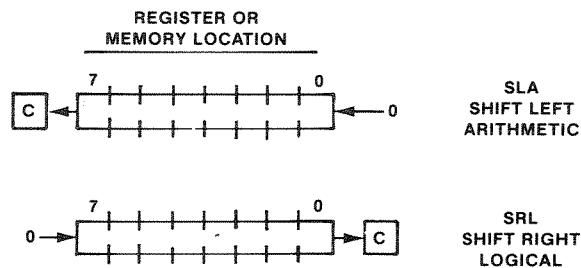


Figure LESS20-1. Logical Shifts

The two logical shifts in the Z-80 are the SRL (Shift Right Logical) and SLA (Shift Left Arithmetic). Even though the second mnemonic has an “arithmetic” modifier it is still a logical shift!

As in the case of the rotates, any bit leaving the register or memory location as a result of the shift goes into the Carry flag. The flags are set as in the case of the “any register” rotates, with C, Z, P/V, and S being affected.

The SRL instruction shifts one bit right, filling a zero into bit 7. If we had 01110001 in the B register, then a

SRL B ;shift B right

would result in a 00111000 with the Carry flag set to 1. Eight SRLs would result in 00000000, clearing the B register.

The SLA instruction works exactly the same way in reverse. If we had 01110001 in B, then

SLA B ;shift B left

would result in 11100010, with the Carry flag set to 0.

SRL and SRA can be used with any register and with (HL) or indexed addressing modes.

Multiplying and Dividing By Shifting

Every time a logical shift left is done, the original value in the register or memory is multiplied by 2:

	00110010	Original=50
←	01100100	After SRL=100
←	11001000	After SRL=200 (absolute)
←	10010000	After SRL=144 (invalid)

20 More Shifting and Multiplication

Every time a logical shift right is done, the original value is divided by 2:

	00110010	Original=50
→	00011001	After SRA=25
→	00001100	After SRA=12
→	00000110	After SRA=6
→	00000011	After SRA=3

You can see from the above examples that there is a limit to the number of multiplies by shifting that can be done. This limit is the size of the register or memory location itself. After a certain point, the results are invalid.

Another interesting point is that a divide by shifting to the right results in a loss of the remainder. Actually, a divide by shifting puts the remainder (0 or 1) into the Carry flag, but it is lost on the next shift.

Multiplying and dividing by shifting, then, can be done for small values and with an eye on the limitations of this type of processing.

Multiplication by powers of two can also be done by adding either 8-bit values or 16-bit values to themselves, as we saw in an earlier lesson.

Software Multiplies

What about multiplication by other than powers of two? Suppose we wanted to multiply any 8-bit number by any other 8-bit number? There is no built-in “hardware” multiply or divide in the Z-80. We’ve got to do everything in software! This is not an unusual situation for microcomputers, by the way.

The following program (at the start of the Lesson File) shows one way of implementing an “8 by 8” multiply:

```
100 ;.....
110 ;• EIGHT BY EIGHT MULTIPLY, UNSIGNED      •
120 ;•   ENTRY: E=MULTPLICAND                  •
130 ;•   A=MULTIPLIER                          •
140 ;•   EXIT: (HL)=RESULT                      •
150 ;.....
160 MULTLY   LD      D,0      ;multiplicand now in DE
170         LD      B,8      ;for 8 multiplier bits
180         LD      HL,0     ;initialize result
190 MULO10   ADD     HL,HL    ;shift result
200         SLA     A        ;shift mult'ier left
210         JR      NC,MULO20 ;go if C=0
220         ADD     HL,DE    ;add in mult'cand
230 MULO20   DJNZ   MULO10   ;go if not 8 times
240         RET            ;return
```

This multiply is in the form of a subroutine in case you want to use it in some of your own assembly-language programs.

MULTLY is entered with the multiplicand in E and the multiplier in A. In case you’ve forgotten, the multiplicand is the number on the top and the multiplier is the number on the bottom in a multiply. It really doesn’t make any difference which number you put where in this case, however.

At the end of the multiply, HL holds the result (the product).

Assemble the program.

Execute the program after first using ZR commands to load the E and A registers with two numbers to multiply. Now an important point: Breakpoint the RET instruction by using the ZB command as in

ZB XXXX (BREAKPOINTS THE RET AT LOCATION XXXX)

If you don't breakpoint the RET, the RET will try to execute, and cause either a 'NOT AN INSTRUCTION' or 'JP, REF, OR STK OUT OF OBJECT' error.

Record the numbers and results for these number combinations:

- 1) 50H times 02H (80 * 2)
- 2) 0AH times 14H (10 * 20)
- 3) 00H times 00H (0 * 0)
- 4) 7FH times 02H (127 * 2)
- 5) 80H times 02H (128 * 2)
- 6) FFH times FFH (255 * 255)

Got them? Let's do some analysis of the multiply. We're multiplying an 8-bit number by an 8-bit number. How large will the result be? That's fairly easy to figure out.

We know that in 8 bits we can hold 0 through 255 if the numbers are "unsigned" or absolute. The result of the multiply can therefore be anything from 0 (0 * 0) through 65,025 (255 * 255).

Since a 16-bit number can hold 0 through 65,535, it looks as if we should have no trouble fitting the quotient in the 16-bit HL register pair. Here is what you should have seen after the multiplies:

- 1) 50H times 02H=00A0H (80 * 2)=160
- 2) 0AH times 14H=00C8H (10 * 20)=200
- 3) 00H times 00H=0000H (0 * 0)=0
- 4) 7FH times 02H=00FEH (127 * 2)=254
- 5) 80H times 02H=0100H (128 * 2)=256
- 6) FFH times FFH=FE01H (255 * 255)=65,025

It appears that the multiply works, but how is it done?

This multiply, and many software multiplies, emulate paper and pencil multiplies. Suppose that we take the case of 10 times 20. We can do binary multiplication with paper and pencil by the process shown in Figure LESS20-2.

<pre> 00001010 = 10 00010100 = x 20 ----- 00000000 00000000 00001010 00000000 00000000 00001010 00000000 00000000 00000000 00000000 ----- 000000011001000 = 200 </pre>	<p>METHOD 1</p>
<pre> 00001010 = 10 00010100 = x 20 ----- 00000000 00000000 00000000 00001010 00000000 00001010 00000000 00000000 00000000 00000000 ----- 000000011001000 = 200 </pre>	<p>METHOD 2 (CLOSE TO A "SOFTWARE" MULTIPLY)</p>

Figure LESS20-2. Paper and Pencil Binary Multiply

20 More Shifting and Multiplication

All we're really doing in the process is shifting over to the next "bit position" and adding in either zeroes, or the multiplicand, depending upon whether the multiplier bit was a 0 or 1.

In MULTLY, we're shifting the result in HL to the left instead of shifting the multiplicand as we add it in. The shift is performed by the method of adding, which shifts HL one bit position left. The next multiplier bit is shifted out from the left of A by an SLA; it sets the Carry.

If the Carry is set after the shift of A, the multiplicand is added to the "partial" result. If the Carry is not set, nothing is added.

If you can't see how this works, and it is hard to visualize, try this: Play computer with paper and pencil. Write down all of the registers, and then go through each instruction. You should wind up with something like Figure LESS20-3.

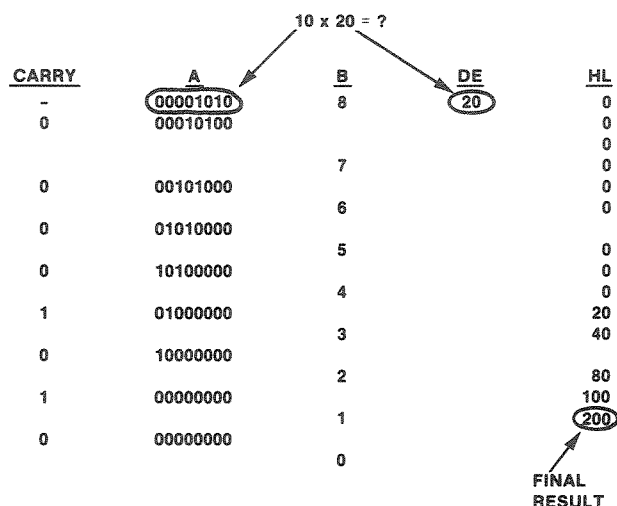


Figure LESS20-3. Playing Computer

Now execute MULTLY again and compare what you've written down with the display at slow speed.

A similar type of multiply can be implemented for "16 by 8" operands, or even "16 by 16" operands. The product will never be larger than the total number of bits in both operands. For example, a 16 by 4 multiply will yield a 20-bit product, maximum.

A 16 by 8 Multiply

Here's another method for a 16 by 8 multiply. Let's say that one operand is in location B000H and B001H in standard Z-80 16-bit format and that the other operand is in location B002H in 8 bits. We want to put the product in B003H on. We'll use the MULTLY subroutine to do the multiply. Can you do it? Think about it for a second, and we'll give you the answer.

One way to do it is to use the fact that any 16-bit number of the form hex XXYYH is really $XX \cdot 256 + YY$. If we call the two bytes of the first number AB and the byte of the second number C, then we have $(A \cdot 256 + B) \cdot C$. This is equal to $A \cdot C \cdot 256 + C \cdot B$, which is the same as

$$A \cdot C \cdot 256 = A \cdot C \text{ shifted left 8 bits} \\ + C \cdot B$$

All we have to do is separate multiplies of $A \cdot C$ and $C \cdot B$ and do some shifting and addition and we'll have the result (note: do not delete lines 100 — 240!):

250 ; 16 BY 8 MULTIPLY BY PARTIAL PRODUCTS

260 MUL16	LD	A,(0B002H)	;get C
270	LD	E,A	;in E
280	LD	A,(0B001H)	;get A
290	CALL	MULTLY	;A*C
300	PUSH	HL	;save
310	LD	A,(0B000H)	;get B
320	LD	E,A	;in E
330	LD	A,(0B002H)	;get C
340	CALL	MULTLY	;C*B
350	LD	A,L	;ls byte
360	LD	(0B005H),A	;store
370	LD	L,H	;2nd byte
380	LD	H,O	;now in HL
390	POP	BC	;get first result
400	ADD	HL,BC	;find 1st, 2nd byte
410	LD	A,L	;get next byte
420	LD	(0B004H),A	;store next byte
430	LD	A,H	;get ms byte
440	LD	(0B003H),A	;store
450	END		;end

Assemble and execute the program using ZX MMMM, where MMMM is the execution address corresponding to location MUL16. Try different operands in B000H/1H and B002H. You'll see the result as a 3-byte number in B003H through B005H. **The number will not be in standard format, but will be ordered from most significant byte to least significant.**

We'll leave it up to you to go through the program in detail.

This technique can be used for 16 by 16 multiplies or even greater, but does get fairly tedious for a larger number of bytes. **Floating-point** representation is generally used for large numbers, as in BASIC single-precision and double-precision variables.

Arithmetic Shifts

We've gotten off on a tangent here discussing multiplies, but we really couldn't do them justice before discussing shifts.

We mentioned another type of shift at the beginning of this lesson called the "arithmetic shift." To see how this shift works, enter the following code and assemble:

460 ; ARITHMETIC SHIFTS

470 ASHFT	LD	A,85H	;load A with 10000101
480	LD	B,8	;loop count
490 ASH010	SRA	A	;shift A right, arith
500	DJNZ	ASH010	;go if not 8
510	END		;end

Execute the program at slow speed and watch the contents of the A register.

What did you see?

You should have seen the A change as follows:

20 More Shifting and Multiplication

```
10000101
11000010
11100001
11110000
11111000
11111100
11111110
11111111
```

Now change the 85H to 75H and execute. You should see:

```
01110101
00111010
00011101
00001110
00000111
00000011
00000001
00000000
```

It appears that in the first case the shift “extends” ones, while in the next case zeroes are “extended.” Why?

The arithmetic shift is used for signed values. When the most significant bit is a sign (or even if it isn’t), the SRA will extend the msb to the right as the number is shifted. For positive numbers (sign bit=0), this works the same as an SRL, extending zeroes. For negative numbers, though, the result is different.

Looking back on the shift of 85H, let’s take the two’s complement of the result and see what we get:

10000101	-123
11000010	-62
11100001	-31
11110000	-16
11111000	-8
11111100	-4
11111110	-2
11111111	-1

Aha! Looks like the SRA can be used to “sign extend” the result. By sign extend we mean that the result will be shifted right with the sign intact. If we used just an SRL shift, we’d have an invalid result as in

10000101	-123
01000010	+66

Again, as in the case of an SRL, we lose a portion of the result on the shift if the number is odd. The -123 became a -62, for example. The SRA is handy, though, for those cases where we want to shift a negative number and do it with the sign properly adjusted.

To Sum It All Up

To recap this lesson:

- The SRL and SLA are “logical” shifts that shift a register or memory location to the right or left one bit at a time
- Logical shifts set the C, Z, S, and P/V flag
- Logical shifts to the right divide by 2 and to the left multiply by 2
- Software multiplies in the Z-80 emulate paper and pencil binary multiplication

- The product of binary multiplies will not exceed the total number of bits in both operands
- The SRA arithmetic shift sign extends the register or memory location contents as the shift is done

For Further Study

Multiple-precision operations (Lesson 11)



Lesson 21

Bit Operations and Divides

Load LESS21 from disk.

In this lesson we'll look at another way of manipulating data at the bit level. The BIT, SET, and RES instructions let us test, set, or reset a bit in a register or memory location. Each of these instructions replaces as many as three "8080" instructions that would accomplish the same task.

The BIT Instruction

The BIT instruction is used to test the state of any bit in a register or memory location. BIT 7,C, for example, will test bit 7 of the C register. The state of the bit will go into the Z flag. If the bit is a 1, the Z flag will be set (Z); if the bit is a 0, the Z flag will be reset (NZ).

As an example, suppose that we wanted to test the state of location B000H, bit 5, and jump to SETJMP if the bit is a 1 or to RESJMP if the bit is a 0. We could do it by:

LD	HL,B000H	;point to location
BIT	5,(HL)	;test bit
JR	NZ,SETJMP	;go if 1
JR	RESJMP	;go if 0

The equivalent code without the BIT instruction would be:

LD	HL,B000H	;point to location
LD	A,(HL)	;get byte
AND	20H	;test bit
JR	NZ,SETJMP	;go if 1
JR	RESJMP	;go if 0

The format of the BIT instruction is

BIT	N,XXXX
-----	--------

where n is the bit number to be tested. Bit numbers are 7 through 0, as we've seen in other examples, with bit 7 on the left and 0 on the right.

The addressing modes permitted in the BIT instructions are:

- Register addressing, as in BIT 4,H
- HL register indirect, as in BIT 5,(HL)
- Indexed addressing, as in BIT 4,(IX+23) or BIT 5,(IY-24)

The SET Instruction

The SET instruction sets a given bit in a register or memory location. To set bit 1 of the 23rd byte from a location pointed to by IY, for example, you'd have

SET	1,(IY+23)	;set bit 1
-----	-----------	------------

The SET replaces as many as three instructions. Without the SET, you'd have something like:

LD	A,(IY+23)	;get byte
OR	2	;set bit 1
LD	(IY+23),A	;restore byte with 1 set

Another advantage of the BIT, SET, and RES is that you don't need to use a register as we did here to manipulate the bit.

Addressing modes are the same as BIT, so you can SET bits in registers or memory locations by instructions such as

21 Bit Operations and Divides

SET	3,L	;set bit 3 of L
SET	4,(HL)	;set bit 4 of (HL)
SET	6,(IX-6)	;set bit 6 of (IX-6)

RES Instruction

The RES instruction resets a bit in a register or memory location. To reset bit 7 of the location pointed to by HL, you'd have

RES	7,(HL)	;reset bit 7
-----	--------	--------------

The equivalent code without the RES would be

LD	A,(HL)	;get byte
AND	7FH	;reset bit 7
LD	(HL),A	;store byte

The addressing modes are the same as SET, so you can have instructions like

RES	0,L	;set bit 0 of L
RES	3,(HL)	;set bit 3 of (HL)
RES	2,(IX+6)	;set bit 2 of (IX+6)

Flags for the BIT, SET, and RES Instructions

The SET and RES instructions don't affect any Flags. The BIT instruction sets the Z flag to the result of the test and leaves the Carry flag unchanged.

A Divide Using SET, RES

To show you how these instructions work, we'll use a divide subroutine to complement the multiply subroutine in the previous lesson.

Enter the following code, or use the Lesson File:

```
100 ;.....
110 ;• DIVIDE 16 BY 8 SUBROUTINE, UNSIGNED •
120 ;• ENTRY: (HL)=16-BIT DIVIDEND •
130 ;• (C)=8-BIT DIVISOR •
140 ;• EXIT: (HL)=QUOTIENT •
150 ;• (A)=REMAINDER •
160 ;.....
170 DIVIDE XOR A ;clear A
180 LD B,16 ;16 iterations
190 DIVO10 ADD HL,HL ;shift residue left
200 ADC A,A ;shift A with carry
210 SET O,L ;Q=1
220 SUB C ;subtract divisor
230 JR NC,DIVO20 ;go if not negative
240 ADD A,C ;restore
250 RES O,L ;reset Q to 0
260 DIVO20 DJNZ DIVO10 ;do 16 times
270 RET ;return
```

Assemble the program. DIVIDE divides a 16-bit number by an 8-bit number. Use the ZR capability of ALT to set up the HL and C registers to different operands. Use the ZB command to breakpoint at the RET instruction so that you can look at the results.

Execute DIVIDE and observe the results. Some typical values might be these:

FFFFH/01H=FFFFH	00H	65,535/1=65,535 remainder 0
F000H/20H=0780H	00H	61440/32=1920 remainder 0
03E8H/75H=0008H	40H	1000/117=8 remainder 64
0003H/30H=0000H	03H	3/48=0 remainder 3

How does the DIVIDE work? Divides are a little more cumbersome than multiplies. The general method used in divides is “restoring division.”

A paper and pencil division is shown in Figure LESS21-1. The divisor (the number that goes “into” the dividend) is tried with the first digit of the dividend. If this doesn’t “go,” the next two digits of the dividend are tried. If this doesn’t work, the next three digits of the dividend are tried.

$$43 \overline{) 23915} = ?$$

	0000001000101100 = QUOTIENT OF 556
00101011	<div style="border-left: 1px solid black; padding-left: 5px; margin-left: 5px;"> 0101110101101011 -0101011 00000111011 -00101011 0001000001 -00101011 000101100 -00101011 0000000111 = REMAINDER OF 7 </div>

Figure LESS21-1. Paper and Pencil Binary Division

If the divisor does “go,” it is subtracted from the dividend. The next digit is then brought down with the result, and the process is repeated.

What we are doing in our heads is to make the determination that the divisor will “go” into the next dividend “residue.” In some cases we actually try a quotient digit and find that the result is too large to be subtracted from the residue; it would give a negative number. In these cases we “restore” the original residue and try again.

When implemented in a software divide, the program **always** subtracts the divisor from the residue, as shown in Figure LESS21-2. If the result of the subtraction is negative, the subtraction won’t “go.” In this case the residue is “restored” by adding back the divisor, and the quotient (result) bit is set to 0. If the subtraction does go, the quotient bit is set to 1 and no “restore” is done.

21 Bit Operations and Divides

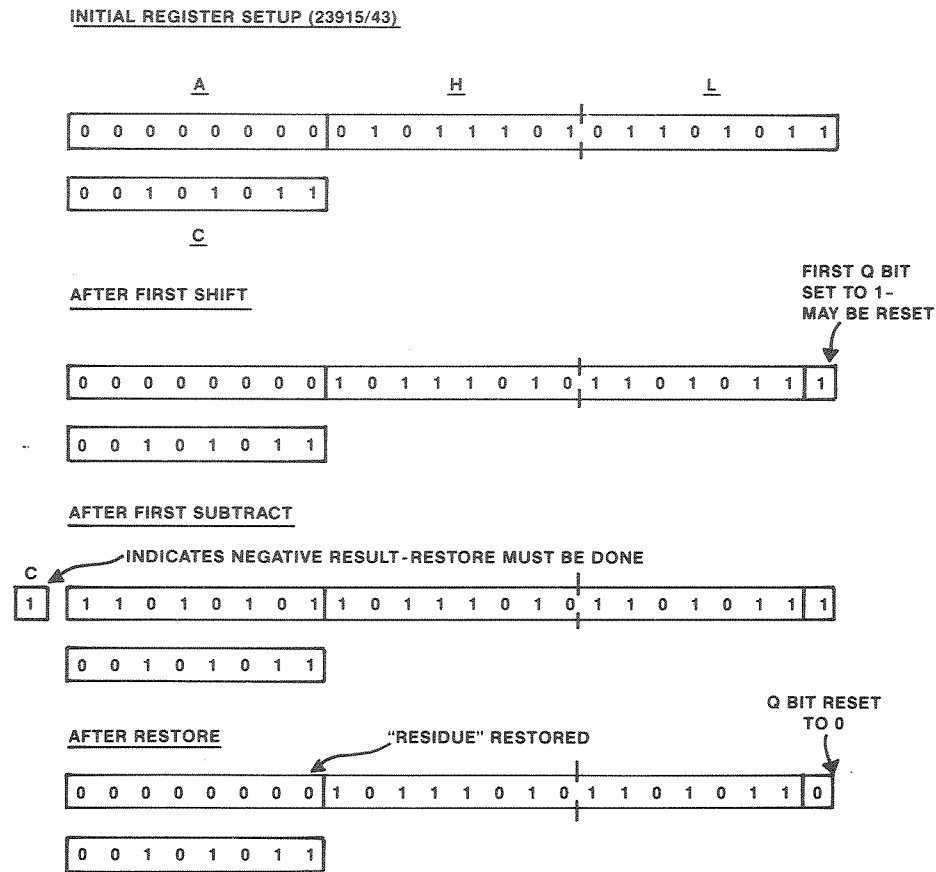


Figure LESS21-2. Software Divide

The **DIVIDE** program shown above uses the **HL** register pair to hold the dividend, as shown in Figure LESS21-2. The **A** register can be considered an “extension” of the dividend, as the dividend residue is shifted into the **A** register for subtracts. The **C** register holds the divisor, and it is subtracted from **A**.

There will be a maximum of 16 bits in the quotient, as in **FFFFH** divided by 1. Therefore there will be 16 subtracts, each one generating a quotient bit of 0 or 1.

The quotient bit goes into the least significant bit of the **L** register. It is **SET** to a 1 before the subtract and **RESet** if the subtract doesn’t go. The quotient bit can be put into this bit because the **HL** register leaves a vacated bit as it is shifted left into the **A** register.

The loop from **DIV010** in **DIVIDE** is the main loop. It first **ADDs** **HL** and **HL** to shift part of the residue and any accumulated quotient bits to the left. This add leaves a bit in the carry from the add which is really the highest order bit from **HL**. This bit is shifted into the **A** register as **A** is shifted by the **ADCA,A**.

Next, the **Q** bit is set to 1.

The divisor in **C** is then subtracted from the residue in **A**. If this residue “goes negative,” the Carry flag will be set. In this case **C** is added back to the residue and the **Q** bit is **RESet** to 0.

The **DJNZ** loops back to the next subtract for the 16 iterations of **DIVIDE**.

On exit, **A** and **HL** have been shifted 16 times and 16 subtracts with possible restores have been done. The quotient is in **HL**, and the remainder is in **A**.

Now that you understand how the **DIVIDE** works, execute it again at slow speed and pay close attention to the results after each subtract in **A**, the shifting of **A** and **HL** and the **SET** and **RES** of the quotient bits.

DIVIDE is a typical divide in software. Like MULTLY, it is an “unsigned” divide that does not work with two’s complement numbers.

There is one case in the DIVIDE which you should be aware of. This is division by zero. If you divide FFFFH by 01H (65,535/1), you’ll get a quotient of FFFFH, which is correct. What do you get when you try dividing FFFFH by 00H? Try it and see. What about 03F8H by 00H?

Both of these cases and **any** division by 0 produces FFFFH, which is incorrect. Because of this, using a 0 divisor is not allowed in divides and is not allowed in most other mathematics operations.

Dividing by Larger Numbers

The DIVIDE above is about the minimum-sized divide that is still useful. By proper “scaling” you can use DIVIDE to get fairly good accuracy. For example, if you wanted to add $1/2 + 1/4 + 1/8$, and so forth, you could do the divide as $10000/2 + 10000/4 + 10000/8 \dots$. The resulting quotient would be “scaled up” to 10,000 times the actual result, and you could find the true result by putting a decimal point four places in front of the computed result:

```

10000/2=   5000
10000/4=   2500
10000/8=   1250
10000/16=   625
10000/32=   312
10000/64=   156
10000/128=   78

```

9921 => .9921 actual is .9921875

As in the case of the multiply, working with divides for larger numbers becomes tedious. It’s relatively easy to program a 32-bit by 16-bit divide which would give a 32-bit maximum product with a remainder of 16 bits and you might want to try this as an exercise. Use a register pair such as BC or DE and put half of the dividend in HL at a time. Use the other register pair to collect the 16 bits of the quotient.

Doing “Signed” Multiplies and Divides

In this lesson and the last we’ve discussed **unsigned** multiplies and divides. How would you do “signed” multiplies and divides?

Although it’s possible to do them directly with special multiplies and divides, the easiest way is to first convert the operands to absolute values, do the multiply or divide, and then change back the result to the proper sign. An example is shown for DIVIDE (don’t delete the DIVIDE program when running this):

21 Bit Operations and Divides

280 ; SIGNED DIVIDE

290 SDIV	LD	HL,(0B000H)	;get dividend
300	LD	A,(0B002H)	;get divisor
310	LD	C,A	;in C
320	LD	A,H	;sign of dividend
330	XOR	C	;find result sign
340	PUSH	AF	;save result sign
350	BIT	7,H	;test sign of dividend
360	JR	Z,SDI010	;go if +
370	CALL	NEGHL	;find absolute value
380 SDI010	LD	A,C	;get divisor
390	BIT	7,A	;test sign
400	JR	Z,SDI020	;go if +
410	NEG		;find absolute value
420 SDI020	LD	C,A	;back to C
430	CALL	DIVIDE	;do unsigned divide
440	POP	BC	;get sign of result
450	BIT	7,B	;test
460	JR	Z,SDI030	;go if +
470	CALL	NEGHL	;negate HL to absolute
480	NEG		;negate A
490 SDI030	LD	(0B003H),HL	;store quotient
500	LD	(0B005H),A	;store remainder
510	JR	DONE	;done
520 NEGHL	EX	DE,HL	;HL now in DE
530	LD	HL,0	;clear HL
540	OR	A	;clear carry
550	SBC	HL,DE	;absolute value of HL
560	RET		;return
570 DONE	END		;end

To see how this works, put a dividend in B000H and B001H and a divisor in B002H. The quotient is stored in B003H and B004H and the remainder in B005H. Execute from "SDIV"

The absolute value of HL is taken by clearing it to 0 and then subtracting the original value of HL from 0. The absolute value of A is taken by a NEGate instruction.

The trick here is to get the sign of the result. The sign of a multiply or a divide is the exclusive OR of the two operands:

+ TIMES A + = +	0 XOR 0 = 0
+ TIMES A - = -	0 XOR 1 = 1
- TIMES A - = +	1 XOR 1 = 0

(same for divides)

The exclusive OR is taken and the result saved in the stack. Bit 7 of the result value is the sign, 0(+) or 1(-). After the operands have been converted to their absolute values, the sign result is POPped from the stack and used to convert the result of the divide to the proper sign.

The operands initially should be 16-bit or 8-bit two's complement numbers. The results will also be two's complement numbers. Can you detect a case in which the DIVIDE will not work properly?

Did you find the case? If -32768 (8000H) is divided by -1 (FFH), the result of 32,768 is too large to be held in 16 bits. The largest positive 16-bit number is 32,767, or 7FFFH.

To Sum It All Up

To review what we've learned in this lesson:

- The BIT instruction tests a single bit in a register or memory byte and sets the Z flag to the bit state
- The SET and RES set or reset a single bit, respectively
- Flags are not affected by the SET or RES, but are affected by the BIT
- Software divides generally use a “restoring” division technique of shift, subtract, and possible restore
- Numbers can be “scaled up” for multiplies and divides
- Signed multiplies and divides may be done by converting to absolute values, performing the operation, and converting the result to proper sign
- Exclusive ORing two operands for a multiply or divide will give the proper result sign in the sign bit of the result of the XOR

For Further Study

Flag settings for BIT (Appendix V)



Lesson 22

Bits and Pieces

Load LESS22 from disk.

Up to this point we've covered most of the major instructions in the Z-80. The preceding lessons have described instructions that will be used in 95% of your assembly-language code. In this lesson we'll round out the instruction set by briefly mentioning some of the less frequently used instructions.

The Decimal Instructions

In Lesson 19 we briefly discussed binary-coded-decimal (bcd) instructions. You may want to go back and review that material now, because we're going to discuss an instruction that makes bcd additions and subtractions possible, the DAA, or Decimal Adjust Accumulator.

Enter the following program, or use the source code from the Lesson File:

```
100 ; BCD OPERATIONS
110 BCDST      LD      IX,0B00BH      ;point to result
120           LD      HL,0B003H      ;point to BCD op 1
130           LD      IY,0B007H      ;point to BCD op 2
140           OR      A              ;clear Carry
150           LD      B,4             ;loop count
160 BCDO10     LD      A,(HL)         ;get 1s byte
170           ADC     A,(IY)         ;get 1s res, binary
180           DAA                    ;decimal adjust
190           LD      (IX),A          ;store bcd
200           DEC     IX              ;op 1 pointer
210           DEC     IY              ;op 2 pointer
220           DEC     HL              ;result pointer
230           DJNZ    BCDO10          ;go for next
240           END                    ;end
```

This program takes two 4-byte operands and adds them together in a multiple-precision operation. (You may want to review Lesson 11, which talked about multiple precision.) The first 4-byte operand is located in locations B000H through B003H. The second is located in locations B004H through B007H. The bcd result is stored in locations B008H through B00BH.

All operands are treated as 32-bit numbers, with the most significant byte on the left and the least significant byte on the right.

Assemble the program. Now store these values in the operands by using ZM:

Operand 1: 12H, 34H, 56H, 78H
Operand 2: 56H, 78H, 91H, 23H

Execute the program and look at the results.

What did you find? The result of a binary add is predictable and not too hard to figure out, even though we are working with 32-bit numbers:

```
12345678H
+56789123H
-----
68ACE79BH
```

22 Bits and Pieces

What does this number represent in binary? Some ungodly number, no doubt . . . That's not the point — look at the result of the bcd add in locations B008H through B00BH.

The result of the bcd add was:

```
  12345678H
+56789123H
-----
  69134801H
```

In fact, the bcd add enables us to treat the two operands as if they were decimal numbers, and not binary. The result was the same as if we had added the two numbers with paper and pencil.

The decimal adjust accumulator (DAA) takes each binary result and converts it to a bcd-format number by adding 6 to each 4 bits, under certain conditions. It adjusts the result from a binary result to a bcd result, eliminating the invalid bcd 4-bit groups of 1010, 1011, 1100, 1101, 1110, and 1111.

Try some other operands, and you'll see the difference. Of course, you must start off with valid BCD numbers in both of the operands, numbers that have the valid bcd digits of 0000 (0) through 1001 (9) in each digit position.

The DAA automatically handles "carries" also, so that you can work with "multiple-precision" bcd operands, just as you did with multiple-precision binary operands.

The DAA will also work in similar fashion for subtracts. The DAA should follow every 8-bit subtract and will adjust the operand to a bcd result. We'll leave it up to you to reassemble with "SBCs" in place of the ADCs.

BCD numbers take more space, as you can see from the results. Conversions between ASCII and bcd are somewhat simpler, however. To convert from an ASCII character of 30H through 39H, representing "0" through "9," all you have to do is something like:

CONVRT	CALL	GETCHR	;get character
	SUB	30H	;convert to bcd 0 — 9
	RRD		;store in next 4 bits

The SUB above converts the ASCII 30H — 39H to 00H — 09H, and the RRD instruction stores the 4-bit bcd value into the next position in a buffer. See, there was a use for RRD after all!

Using the Second Register Set

We haven't used any of the second set of registers that we talked about early in this course. As you'll recall, AF, BC, DE, and HL have a duplicate set of registers, designated AF', BC', DE', and HL'. Only one set can be current at any time.

On system startup, one set is selected. To switch to the other set, two instructions are used — EX AF,AF' and EXX.

EX AF,AF' switches to the second AF pair. EXX switches to the second pair of BC, DE, and HL.

To see how they work, delete lines 100 — 240 and use the Lesson File:

```

250 ; EX AF,AF' AND EXX
260 EXTST      LD      A,1           ;typical args to regs
270            LD      BC,1234H
280            LD      DE,5678H
290            LD      HL,9ABCH
300            EX      AF,AF'        ;switch to alternate
310            EXX                     ;switch remainder
320            LD      A,2           ;typical args to primes
330            LD      BC,-1
340            LD      DE,-2
350            LD      HL,-3
360            EX      AF,AF'        ;switch back
370            EXX
380            END

```

Assemble this program and execute slowly. You'll first see A through L loaded with typical values. Next the EX AF,AF' and EXX instructions are executed. At this point, the register display will show a different set of values, whatever is in the alternate set. Note that using the EX AF,AF' and EXX didn't move data, they simply switched to the second set. The next group of loads loads this second set, and you'll see the data displayed on the screen.

Finally, the registers are switched back again, and you'll see the data from the first load.

When should the second set of registers be used? Anytime you'd like. They are there for your convenience. About the only problem in using them is that you may get confused about which set is being used if you do too many switches. Many times using one set is sufficient.

The NOP Instruction

What is a NOP? A NOP is just what it says, a "no operation." A NOP is used to delete instructions without the need to reassemble by substituting a NOP opcode (00H) for all bytes of the instruction.

Suppose that you had the following code:

```

LD      A,(IX+1)      ;get byte
INC     IX            ;bump pointer
INC     IX            ;bump again

```

and you found out that the second INC IX was not needed. You could effectively delete the INC IX without reassembling by replacing the DDH, 23H bytes of the INC IX by NOP codes of 00H, 00H. The NOP does not affect any registers or any condition code settings.

HALT Instructions

The HALT instruction is used on other Z-80 systems, but should never be used on the Model I and Model III. The reason for this is that HALT is "hardwired" from the microprocessor to cause the system to reset. It will not assemble on the ALT, so you won't have to worry about it here.

Interrupt-Related Instructions

DI and EI disable and enable "interrupts" respectively. You really won't need either of these in the programs we're doing here (ALT will not assemble them). Occasionally you'll need to use DI and EI in intermediate or advanced assembly-language programming to turn the interrupts off for operations that are "timed" in software, such as cassette operations and timing loops.

There are three interrupt "mode" instructions which you will not need to use unless you are doing

advanced programming applications, and even then most of the interrupt modes are not used in the Model I and III. IM 0, IM 1, and IM 2 will not assemble on ALT.

IN and OUT Instructions

IN and OUT instructions are used to read and write data to an internal or external I/O device, such as the cassette tape logic or RS-232-C interface. You will not need to use these instructions until you're doing intermediate or advanced applications and, in many cases, not even then.

The IN and OUT instructions operate with the A register and move a byte of data to or from an input/output "port" whose address is 0 through 255. There are some port addresses which are dedicated to system functions, such as port 0FFH, which is dedicated to cassette tape operations.

There are a number of input/output "block" instructions which help to move blocks of data on an input or output. Operation is somewhat similar to the block move instructions we discussed earlier in these lessons. Chances are you'll never use any of these, even in advanced programming applications. For reference, the mnemonics are OUTI, OTIR, OUTD, OTDR, INI, INIR, IND, and INDR. None will assemble on ALT.

RST Instructions

The RST instruction includes 8 separate formats: RST 00H, RST 08H, RST 10H, RST 18H, RST 20H, RST 28H, RST 30H, and RST 38H. RST operation is identical to the CALL unconditional. It jumps to location 00H, 08H, . . . , or 38H after first pushing the return address in the stack.

The key to the RST is that it is a **single** byte CALL of the format 11TTTT111, where T is a code of 000 through 111. The address for the jump is found by T*8. If the code were 101, for example, the RST would be 11101111, or an RST 28H (RST 40).

The RST is used all the time in the Model I and III BASIC interpreter, as it makes use of subroutines in the "page 0" (locations 0 through 255) part of memory. RST is only good for CALLing locations 00H, 08H, etc., and for that reason it will be of no use to us here in ALT and very little use in your other programs, unless you are calling some of the BASIC interpreter ROM subroutines.

RST will not assemble on ALT.

Exchange (SP) Instructions

We haven't discussed three instructions which work with the "top of stack." These are EX (SP),HL; EX (SP),IX; and EX (SP),IY. These instructions work like this: The SP points to the last byte stored. The EX takes the last two bytes stored in the stack (SP) and (SP+1) and swaps them with the contents of either HL, IX, or IY.

The instructions are not used for any specific purpose except to enable you to get at stack data without too much manipulation. All will assemble and execute on ALT.

What Do You Do With All the Instructions?

You've covered a lot of ground in the past 22 lessons, and there are a lot of instructions that you have at your command. After programming for a while, you'll come to know them intimately and treat them as old friends. Remember one guiding rule in working with the instruction set: **There is not necessarily a right way to do things.** Many times the same program can literally be implemented hundreds of different ways. Feel free to experiment and try new approaches. You can't go too far wrong. Assembly language is so fast that things will still move swiftly.

To Sum It All Up

To review what we've learned here:

- The DAA does a decimal adjust of the A register after an add or subtract
- The second register set may be activated by EX AF,AF' and EXX; the former selects AF', and the latter BC', DE', and HL'
- NOP is a "do nothing" instruction primarily used for patching
- The HALT should not be used on Model I/III systems
- RST is a special one-byte CALL that is not used unless you are CALLing ROM subroutines in the BASIC interpreter

For Further Study

DAA, NOP, and RST Flags (Appendix V)



Lesson 23

Interfacing to BASIC — Linkages

Load LESS23 from disk.

In previous lessons we've really presented all of the instruction set of the Z-80 microprocessor used on the Model I and Model III. In this lesson we'll assume that you are now an expert on the instruction set (or will be after a mite more study), and we'll concentrate on practical uses of assembly language.

One of the best ways to learn assembly language is to interface it with BASIC in short, high-speed subroutines that complement the flexibility of BASIC. In the next few lessons we'll learn how to do that.

Memory Map

First we'll have to get a clear idea of where we can put assembly-language subroutines. Look at Figure LESS23-1. It shows the general memory layout of a Model I and Model III. Some of it may be familiar to you.

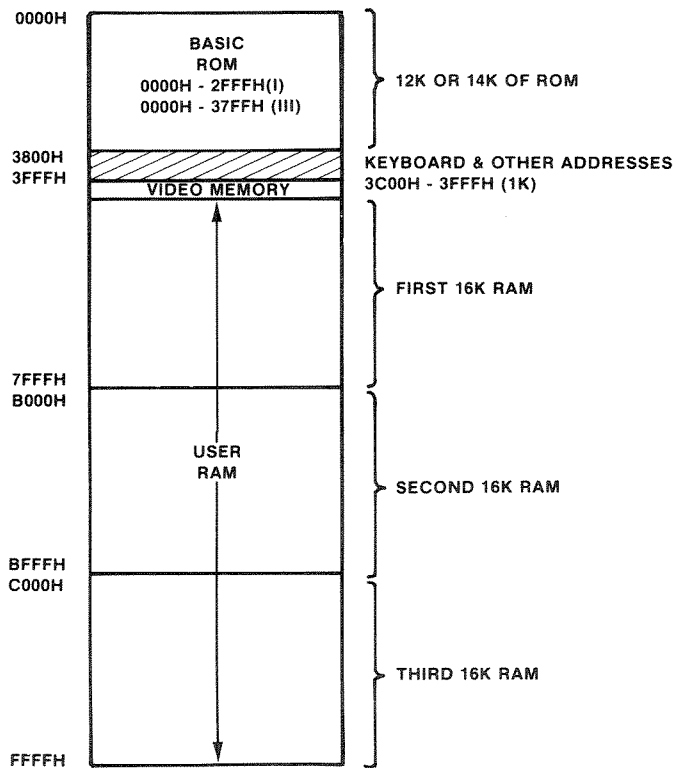


Figure LESS23-1. Memory Layout

The Level II or Model III BASIC interpreter is "burned into" memory locations 0000H through 2FFFH (37FFH in the Model III). This is the "ROM" or "Read Only Memory" portion of the 64K (65,536) bytes of memory available to the largest system.

The memory addresses from 3800H through 3FFFH are "dedicated" to the keyboard and video display.

The keyboard is actually "hardware logic" that looks like a memory device; it is addressed as memory locations 3801H (row 0), 3802H (row 1), on up to 3880H (row 7).

The video memory occupies memory locations 3C00H through 3FFFH. Video memory is very similar to normal RAM (random access memory), except for systems that are upper case only. Although some

23 Interfacing with BASIC — Linkages

special decoding logic is used, in general these locations can be addressed as normal memory locations. To write an ASCII “A” at the upper left-hand corner of the screen, for example, you’d simply do something like this:

```
LD      A,65      ;ASCII A
LD      (3C00H),A ;store
```

The area from 4000H (16K) to 0FFFFH (64K-1) is available for RAM memory. Of course, if you have a 16K RAM system the “top of memory” is at 7FFFH, if you have a 32K RAM system the top of memory is at 0BFFFH, and if you have a 48K RAM system the top of memory is at 0FFFFH.

If you are running a pure assembly-language program without using any BASIC interfacing, then you will have all of the RAM from 4000H through top of memory for your use. (As a practical matter, though, even assembly-language programs will use TRSDOS disk I/O programs at the start of this 4000H area.)

If you are running a combination BASIC and assembly-language program, you’ll find that parts of the RAM area are used as storage by BASIC, by a “resident” section of TRSDOS code, by the BASIC program lines, by BASIC variables, arrays, and strings, by system stack, and by the string storage area. The general scheme is shown in Figure LESS23-2.

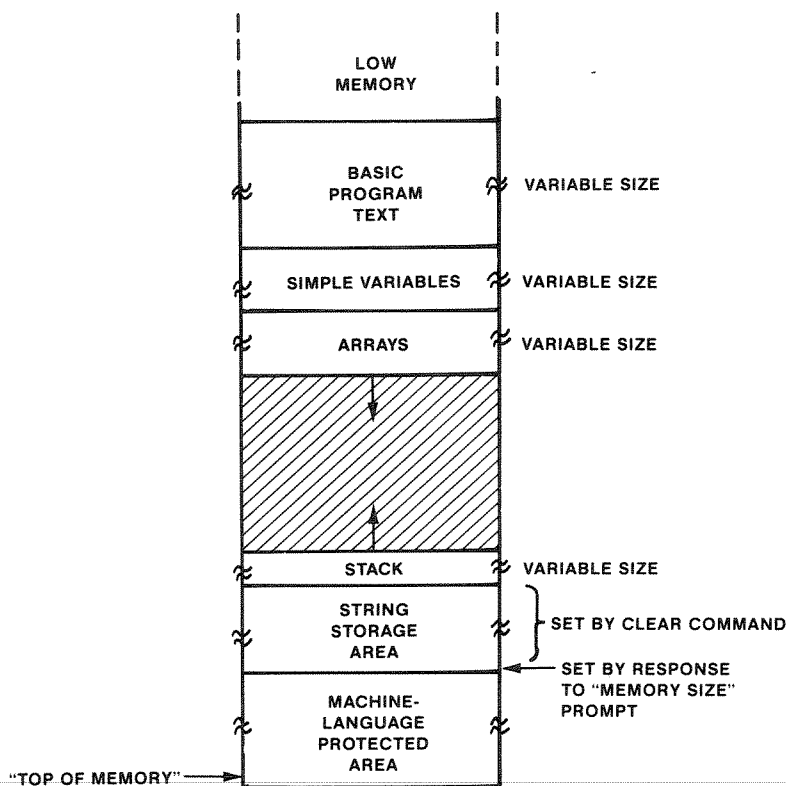


Figure LESS23-2. Using Assembly-Language with Basic

The best place to put a short assembly-language program is as close to the top of memory as possible. You may protect memory used for assembly-language programs by entering a memory value for the MEMORY SIZE? prompt when you first load BASIC.

If you enter 32767 for the MEMORY SIZE? prompt, for example, you’d protect all of memory from 32,768 (8000H) on. BASIC would not use any of that area, and you could use it for assembly-language programs or for any other operations.

To make it easier for you to generate ALT programs and incorporate them into your BASIC system, we'll adopt the following standard for the BASIC interface programs that will be in the following lessons:

Protect the area from 0B800H on for running the following BASIC/assembly-language programs. Enter 47103 in response to the MEMORY SIZE? prompt when loading or reentering BASIC. All of the ALT programs will run in memory from location 0B800H up to top of memory.

Another rule to use: **All programs should be loaded into memory with BASIC at the same area in which they assemble with ALT.** As an example, suppose that you load a program from a Lesson File and the object code from the ALT listing starts at location B800H. You **must** load the object code at location B800H to have it run properly.

THE ORG (Origin) Command

All of the programs we've worked with up to this point have started at the end of the text buffer. In other words, ALT assembles the object code starting at the first location after the text. However, we can use an optional assembler pseudo-op to determine the object start. It is called ORG, for ORiGin, and has the format

ORG XXXXH,

where XXXXH is a hexadecimal starting location.

The origin may be anywhere in RAM that you'd like, as long as it is greater than the end of the text area and less than the top of memory. If you go out of these limits, ALT will let you know.

A practical starting point would be somewhere around B800H. This would be out of the text area for most small programs and far enough down so that it would not interfere with the "symbol table" and other assembler areas.

A map of ALT usage is shown in Figure LESS23-3. It shows you that during an assembly a symbol table of labels is "building down" (from top of memory-256), while at the same time the object code and a "data map" of the type of instruction is "building up." The optimum Origin would be just beyond the text area to avoid problems with lack of space for the symbol table.

23 Interfacing with BASIC — Linkages

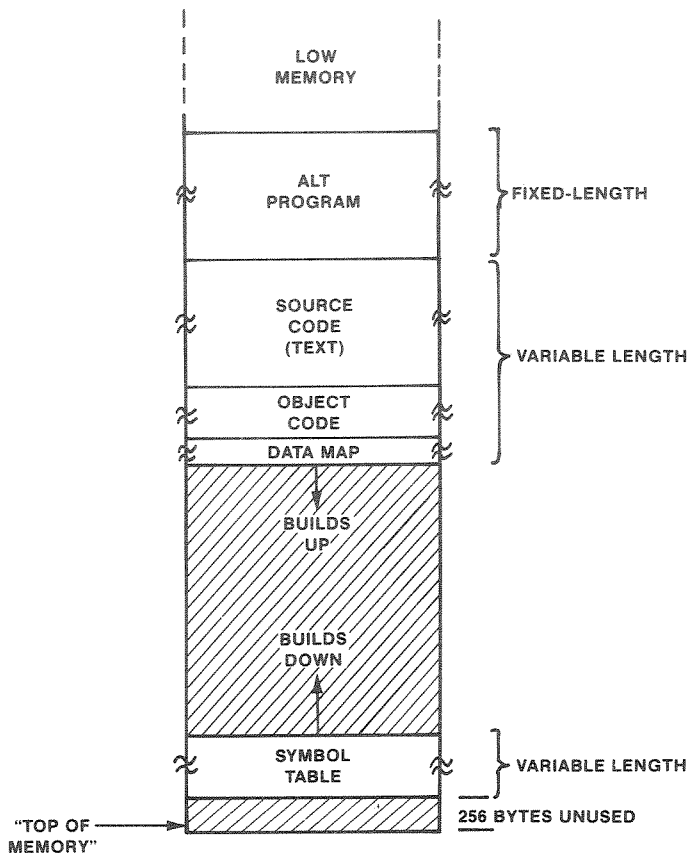


Figure LESS23-3. ALT Memory Usage

To show you how ORG works, enter the following program, or use the Lesson File:

```
100 ; SAMPLE USE OF ORG
110 START      ORG          OB600H      ;assemble at B600H
120            LD          A,(NEXT)     ;load A
130            JR          CONT        ;jump around data
140 NEXT       DEFB        23          ;data
150 CONT       END
```

Assemble the program. Note that the object code starts at location B600H. Jot down the object code or do an ALP listing of the object code by assembling with your system printer. (You must not do a "ZX" or "ZXS" here, as you will get a 'NOT AN INSTRUCTION' message. Execute directly from the START instruction.)

Now delete lines 100 through 150 and look at the next code on the Lesson file:

```
160 START1     ORG          OB700H      ;assemble at B700H
170            LD          A,(NEXT)     ;load A
180            JR          CONT1       ;jump around data
190 NEXT1      DEFB        23          ;data
200 CONT1      END
```

Assemble this segment. Did you notice any difference in the object code?

Relocatability

The assembly at ORG B600H, had

B600	3A 05 B6	START	LD A,(NEXT)
B603	18 01		JR CONT
B605	17	NEXT	DEFB 23

The address of B605 in the LD instruction refers to the “absolute” location of B605H.

The assembly at ORG B700H had:

B700	3A 05 B7	START1	LD A,(NEXT1)
B703	18 01		JR CONT
B705	17	NEXT1	DEFB 23

Even though both LD As referred to a location 3 bytes away, they used different addresses.

Absolute addresses in JPs, LDs, and CALLs, among others, are the reason that object code cannot simply be moved to a new location and execute properly. A reassembly must be done.

Some instructions are “relocatable,” though. An ADD A,C will work anywhere it is located. A relative jump also works anywhere, as it contains no absolute address, but simply a displacement from the current program counter. (Look at the JR instruction for the two assemblies.)

Transferring Control to an Assembly-Language Program

Just how do you transfer control to an assembly-language program from BASIC? There are three steps:

1. Loading the object code of the assembly-language program into RAM
2. Defining where the object code is to the BASIC interpreter
3. Transferring control to the assembly-language program by a USR call in BASIC.

To show you how this process works, let’s use the following program. It is a very simple program to clear the video display. (Never mind that there is a CLS command in BASIC; this time we want to do it ourselves!)

Enter the following source code, or use the Lesson File:

```

210 ; CLEAR SCREEN PROGRAM
220          ORG          0B800H          ;origin
230 CLRSCN   LD           B,20H          ;blank
240          LD           HL,3CO0H        ;start of screen
250 CLRO10   LD           (HL),B          ;store blank
260          INC          HL              ;bump pointer
270          LD           A,H             ;get ms byte
280          CP           40H             ;test for end+1
290          JR           NZ,CLRO10        ;go if not end+1
300          RET              ;return to BASIC
310          END              ;end

```

Assemble the code. You should get an assembly similar to Figure LESS23-4. The important point is that the object code should be identical.

23 Interfacing with BASIC — Linkages

OBJECT CODE SHOULD BE THE SAME IN YOUR PROGRAM

B706		00210 : CLEAR	SCREEN	PROGRAM
B706		00220	ORG	0B800H
B800	06 20	00230 CLRSCN	LD	B,20H
B802	21 00 3C	00240	LD	HL,3COOH
B805	70	00250 CLR010	LD	(HL),B
B806	23	00260	INC	HL
B807	7C	00270	LD	A,H
B808	FE 40	00280	CP	40H
B80A	20 F9	00290	JR	NZ,CLR010
B80C	C9	00300	RET	
B80D		00310	END	

Figure LESS23-4. CLRSCN Assembly

Loading the Object

In a Series I Editor/Assembler we could load the object code into RAM by using cassette tape or disk. In our case, however, we'll use a different technique. We'll load the object by using the POKE statement in BASIC.

In case you're not familiar with the POKE, it works like this:

```
POKE 47104-65536,21 'POKE 21 INTO 0B800
```

The POKE statement above stores a decimal 21 into RAM location 47104 decimal, which corresponds to B800H. When working with memory addresses over 32,768 (8000H), the "-65536" is necessary because of a quirk of the BASIC interpreter.

There's only one problem with the POKE. It works with **decimal** values, which means that we have to convert the **hexadecimal** values from the assembly-language listing to decimal before we can use the POKE! You won't find this too much of a chore, however, as we've given you an equivalence table in Appendix III. Also, for every program in these lessons we'll give you the actual decimal values along with the hexadecimal values.

The hexadecimal values for the program (from the listing) are 06H, 20H, 21H, 00H, 3CH, 70H, 23H, 7CH, FEH, 40H, 20H, F9H, and C9H.

The corresponding decimal values (from Appendix III) are 6, 32, 33, 0, 60, 112, 35, 124, 254, 64, 32, 249, 201.

How do we get these values into RAM? An easy way in BASIC is to put the values into DATA statements and then POKE them into RAM with a short loop:

```
100 DATA 6, 32, 33, 0, 60, 112, 35, 124, 254, 64, 32, 249, 201
110 FOR I=47104 TO 47104+12
120 READ A
130 POKE I-65536,A
140 NEXT I
```

In this loop a READ command gets the values from the DATA list. The POKE then POKES the value into the current memory location (don't forget that 47104 corresponds to hexadecimal 0B800H, the ORG point of the program).

The "12" in the FOR...TO statement corresponds to 1 less than the size of the program. If we had a program of 100 bytes, we'd use "40960+99."

We'll assume that we've run the BASIC code above and that the object program has been loaded by the POKES. At this point RAM locations 0B800H through 0B80CH contain the CLRSCN program in "machine language." Nothing mysterious here; we've been doing the same thing in our ALT programs except that we weren't going to run BASIC along with the program. The next step is to tell the BASIC interpreter where the program is.

Defining Where the Object Is to BASIC

If you are running Model I Disk BASIC or Model III Disk BASIC, the following statement will tell Disk BASIC where CLRSCN is located:

```
150 DEFUSRO=&HB800
```

This statement assigns the code of “0” to the CLRSCN location of 0B800H. We could just as well have used DEFUSR3=&HB800, assigning a code of 3.

Transferring Control to the Machine-Language Code

We’re all set up now to transfer control to CLRSCN. We’ll do it by a USR call. The BASIC USR call gets the location from the DEFUSR variable and simply does a CALL. That pushes the return address to the BASIC interpreter in the stack and saves the return point. The last statement in our CLRSCN program is a RET, which pops the stack and causes a return to the BASIC interpreter.

If you have a Disk BASIC, this statement is the one to use:

```
160 A=USRO(0)
```

Executing CLRSCN

We’ve combined all of the statements above into a BASIC program that will:

1. Move the CLRSCN values from DATA statements into the 0B800H area
2. Define the location to BASIC
3. Transfer control to CLRSCN

If you execute the program below, you should see the screen clear in a flash. Load the program by going to BASIC and doing a LOAD “CLRSCN” from disk. Before you do, however

```
PROTECT MEMORY BY ENTERING 47103 FOR THE
MEMORY SIZE?
PROMPT!
```

Now execute the program by RUN.

After the screen clears, you can interrupt the BASIC program by pressing BREAK.

```
100 DATA 6, 32, 33, 0, 60, 112, 35, 124, 254, 64, 32, 249, 201
110 FOR I=47104 TO 47104+12
120 READ A
130 POKE I-65536,A
140 NEXT I
153 DEFUSRO=&HB800
163 A=USRO(0)
170 GOTO 170
```

To get back to ALT, you must reload, as BASIC and ALT are mutually exclusive.

To Sum It All Up

To review the considerable material we’ve covered here:

- Assembly-language programs can be anywhere in RAM when run without BASIC interface
- Assembly-language programs should be in high RAM when run with BASIC and this area should be protected by responding to the MEMORY SIZE? message with the start address of the program-I

23 Interfacing with BASIC — Linkages

- The ORG pseudo-op establishes the assembly origin for a program
- JPs, CALLs, some LDs and other instructions may not be “relocatable”; they often contain absolute addresses that prevent them from running anywhere in memory
- Loading object code can be done by POKEing in BASIC after first converting hexadecimal values to decimal
- The location of the assembly-language program must be defined to the BASIC interpreter by using a DEFUSR statement (Disk BASIC)
- A USR call transfers control to the assembly-language program

For Further Study

BASIC DEFUSR in Disk BASIC (BASIC manual)
BASIC USR or USRn command (BASIC manual)

Lesson 24

Interfacing to BASIC — Passing Parameters

Load LESS24 from disk.

We've covered quite a bit of ground in the previous lesson. Let's review what we did.

First of all we assembled an assembly-language program using ALT. This program took the form of a "subroutine," which, as we know, is really any program that is terminated by a RET, to return to a calling program.

The program was assembled at location 0B800H by using an ORG statement. We aimed for this area because we knew that we were going to be calling the program from BASIC and wanted to locate the program in high enough memory to protect it from overwriting by BASIC statements, variables, and other data.

After the assembly, we didn't load the program directly into RAM by loading a cassette or disk object file, although we could have if we had been using a Series I assembler.

Instead, we converted the hexadecimal values from the assembly listing into decimal values by using Appendix III.

We then made up a BASIC program that consisted of three parts:

1. A DATA statement, or several DATA statements, that had all of the machine language as decimal values.
2. A short FOR . . . TO loop to move these values from the DATA statements into RAM at the 0B800H area
3. A BASIC statement that defined where the assembly-language program was by DEFUSR0=47104
4. A BASIC call to the assembly-language program by a USR0 statement.

We then executed the BASIC program. The program moved the machine-code values from the DATA statements into the 0B800H area. It then took the address of the assembly-language program from the DEFUSR statement and transferred control by a simple CALL instruction, somewhere in the BASIC interpreter.

The CLRSCN program then executed without any BASIC interference. It's important to note that once the assembly-language program is entered, BASIC has no control over it! That's good and bad — if the program has errors in it, there's no easy way to recover, as it may have destroyed critical memory locations used by BASIC.

On the other hand it lets us execute the assembly-language code very rapidly, to supplement BASIC processing.

The last instruction in the CLRSCN was a RET. The RET did not perform magic, but only popped the return address from the stack and caused a return to the BASIC interpreter at some internal point. The "internal point," by the way, is a set of BASIC interpreter code that handles the USR call to assembly-language programs.

The stack used here is an internal BASIC stack, and we don't have to be concerned about establishing our own stack area. There's enough room in the BASIC stack for just about everything we'd want to do in simple assembly-language code. Besides that, if we used our own stack, we'd never be able to return to BASIC unless we carefully saved the return address by something like a

POP	HL	;get return
LD	(RETPT),HL	;save for return

Passing a Parameter

It's easy to see how the DEFUSR works, but what about the USR call? We used the format

```
100 A=USRO(0)
```

What is the (0) and the A variable?

The way that BASIC works is that it takes the value from inside the parentheses, assumed to be a 16-bit integer value, and stores it in a special variable inside BASIC. This value can be accessed by the assembly-language program.

For example, if we wanted to “pass” a value of 223 instead of 0, we'd say

```
100 A=USRO(223)
```

If we wanted to pass the value of a variable, we'd say

```
100 A=USRO(B)
```

We could even pass the value of an expression, as in

```
100 A=USRO(ZZ/256)
```

The only requirement for the value is that it would have to be a BASIC “integer” value of -32768 to +32767. In fact, though, we can fool the BASIC interpreter into accepting an absolute value of 0 through 65,535 by using the following rules:

1. If the value is less than 32768, use the value alone:

```
100 A=USRO(30000)
```

2. If the value is equal to or greater than 32768, use this form

```
100 A=USRO(40000-65536)
```

You've seen in previous programs why we want to pass “parameters” to subroutines. In the SCANTY subroutine of Lesson 18, for example, we passed pointers to a table and the size of the table. Parameters make the subroutine more flexible and generalized.

The USR call, then, lets us pass one 16-bit value as a parameter to the assembly-language subroutine.

Passing a Parameter Back

What about going the other direction? Just as you might suspect, BASIC also allows us to pass a 16-bit value **back** from the assembly-language subroutine to BASIC. You might use the assembly-language code, for example, to scan a table for a certain value and pass back the location of the value, if found.

The A variable in

```
100 A=USRO(B)
```

is set equal to the 16-bit value returned by the assembly-language subroutine. Of course, if the assembly-language subroutine doesn't need to return a value, then A is not used and is a “dummy,” just as the 0 value was in

```
100 A=USRO(0)
```

A Sample Parameter-Passing Subroutine

To show you how this works and what we must do in the assembly-language subroutine, enter the following program, or use the Lesson file.


```

100          ORG          0B800H
110 ;*****
120 ;• SUBROUTINE TO FIND SQUARE ROOTS                      •
130 ;•   ENTRY: SQUARE IN 16 BITS                          •
140 ;•   EXIT:  SQUARE ROOT IN 16 BITS                      •
150 ;*****
160 SQROOT    CALL        0A7FH          ;get argument
170          LD           A,-1          ;clear square root
180          LD           BC,-1        ;initialize odd integer
190 SQRO10    ADD          A,1          ;square root+1
200          ADD          HL,BC        ;subtract
210          DEC          BC          ;BC-1
220          DEC          BC          ;BC-1
230          JP           C,SQRO10     ;loop if not minus
240          LD           L,A          ;square root now in L
250          LD           H,0          ;now in HL
260          JP           0A9AH        ;return argument
270          END           ;end

```

Assemble the program and get a printed listing if you have a system printer. Otherwise note the object code values at the assembly.

This program is a modification of the square root program found in Lesson 9. Refer to that lesson if you like to refresh your memory about how it works. The parameter on entry is a 16-bit square from 0 through 65,535 (unsigned). On exit, the square of 0 through 255 is found. The square is taken to the next lower integer for fractional squares.

If you refer back to the program in Lesson 9, you'll notice two additional instructions.

The CALL 0A7FH instruction at the very beginning is a CALL to a BASIC interpreter subroutine. The BASIC subroutine finds the argument from the USR call and puts it into the HL register pair and then returns to the assembly-language code. This is the way that BASIC passes that 16-bit value from the USR call. At the LD A,-1 instruction, therefore, we've got the argument from BASIC in HL.

The JP 0A9AH instruction jumps back to a BASIC routine that takes the contents of the HL register and stores it into the variable used in the USR call to the left of the equals sign. If we had

```
100 ZZ=USRO(B)
```

for example, variable ZZ would contain the square after the return to BASIC. Note that if **no argument** is to be returned, a RET instruction is used. Only if the argument in HL is to be returned is the JP 0A9AH used in place of the normal RET.

It seems, then, that the entry to assembly language is with an optional argument in HL and that the exit from assembly language is by an optional argument in HL.

Running the Subroutine in BASIC

Ok, it's about time for you to make your first conversion to BASIC. Look at the object code for SQROOT and convert it to decimal values. Then write a BASIC program to store the decimal values in DATA statements, define the location of SQROOT, and call SQROOT. About the only thing you'll have to do differently from the subroutine of the last lesson is to put in some logic for reading the square and listing the square root. Make it as simple as possible.

Got it? See how it compares to this BASIC program:

24 Interfacing with BASIC — Passing Parameters

```
100 REM SQUARE/SQUARE ROOT
110 CLS
120 DATA 205, 127, 10, 62, 255, 1, 255, 255, 198, 1, 9, 11, 11
130 DATA 218, 8, 184, 111, 38, 0, 195, 154, 10
140 FOR I=47104 TO 47104+21
150 READ A
160 POKE I-65536,A
170 NEXT I
183 DEFUSRO=&HB800
190 INPUT SQ
203 SR=USRO(SQ)
210 PRINT "SQUARE=";SQ,"SQUARE RT=";SR
220 GOTO 190
```

Even if your BASIC code does not look quite the same, the DATA values should be identical.

You can now go to BASIC and load the BASIC SQROOT program above. The name is SQROOT. Don't forget to protect memory by responding to the MEMORY SIZE? prompt by 47103! Or, if you'd like, try your own program first.

The program should print the square roots of all numbers input. If the SQ (square) is greater than 32,767, however, you will have a problem. BASIC will accept the entry on the INPUT statement, but when it comes time to make the USR call, BASIC will see that the value is larger than the maximum possible for integer values of +32,767. Do you know how to fix it?

If you guessed that you'd have to use

```
203 SQ=USRO(SQ-65536)
```

in place of the previous statements you'd be partially correct. This will work for all values greater than +32,767, but will not work for values of 0 through +32,767. What we really need is

```
199 SX=SQ: IF SQ>32767 THEN SX=SX-65536
203 SR=USRO(SX)
```

We'll leave it up to you to change the program accordingly.

As SR is returned as a value from 0 to 255, there is no problem with it.

To get back to ALT, you must reload, as BASIC and ALT are mutually exclusive.

To Sum It All Up

To recap what we've learned in this lesson:

- It's not necessary to use your own stack when interfacing to BASIC; BASIC maintains its own
- The USR0() call passes the argument within the parentheses if a "CALL 0A7FH" is done in the assembly-language program
- The USR0 call returns the argument from the assembly-language program if a "JP 0A9AH" is done in lieu of the RET in the assembly-language program
- Variables are passed in the HL register pair

- Variables must be 16-bit integer variables
- If variables are over 32,767, then the XXXXX-65536 form must be used to fool the BASIC interpreter

For Further Study

BASIC integer variables (BASIC manual)



Lesson 25

VARPTR and Passing Multiple Arguments

Load LESS25 from disk.

In this lesson we'll learn some further tricks about how to interface to assembly-language subroutines from BASIC. First of all we'll go into some detail on VARPTR, a BASIC function that is used quite frequently in finding addresses of data to pass to assembly-language subroutines. Secondly, we'll look at how we can pass **multiple** arguments.

VARPTR

VARPTR lets us find the address of any variable in a BASIC program. This is important to assembly-language subroutines because it allows the subroutine to access BASIC data such as arrays and strings.

The format of VARPTR is

100 VARPTR(XX)

where XX is a variable name.

Although VARPTR can be used to find the addresses for integer, single-precision, double-precision, string, and array variables, we'll just be considering string and array variables here. The reason is this: we can already pass an integer-sized value by the USR call, and the single-precision and double-precision variables use a complex floating-point format that is beyond the scope of this text. The VARPTR is most often used with strings and arrays.

Using VARPTR With Strings

When VARPTR is used with a string variable, it has the format of

100 A=VARPTR(AA\$)

where AA\$ is any string variable name.

VARPTR will put the address of a string "descriptor block" in the A (or other) variable. The string descriptor block is shown in Figure LESS25-1.

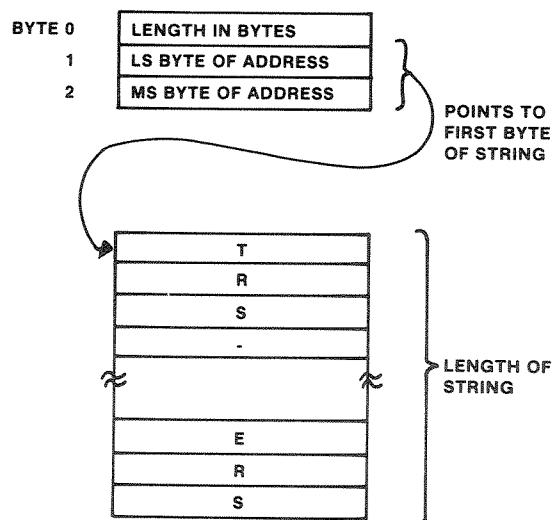


Figure LESS25-1. String Descriptor Block

The first byte of the string descriptor block is the length of the string in bytes. Each character in the string occupies one byte and the total number of string bytes may be 0 to 255.

25 VARPTR and Passing Multiple Arguments

The second and third bytes of the string descriptor block define the actual address of the string. This address is in the standard Z-80 address format we've grown to know and love so well, least significant byte followed by most significant byte.

Where are strings located?

If you have a string in a BASIC statement, such as

```
100 A$="THIS IS A STATEMENT STRING"
```

then the string will be in the BASIC program statement itself. BASIC statements start in low RAM before variable and other storage and continue to build upward. There's a good chance that the BASIC statement and string will be before the 8000H (32768) location and you won't have to use the XXX-65536 format, unless you have a lengthy BASIC program.

```
100 A$="THIS IS A PRO"  
110 B$="CESSED STRING"  
120 C$=A$+B$
```

If you have a string that is "processed," such as C\$ in the above, then the string will be found in the string storage area. This is a temporary storage area for strings that are not present in BASIC program lines. The string storage area is in high memory, just below the protected area for assembly-language subroutines and a BASIC stack. There's a good chance you'll have to use the XXX-65536 format here.

To pass the location of a string to an assembly-language subroutine, you'd have to do something like:

```
100 SL=VARPTR(ZX$) 'FIND STRING DES BLOCK LOCATION  
110 IF SL>32767 THEN SL=SL-65536  
120 A=USRO(SL)
```

Of course, we've left off all of the other logic concerned with moving the machine-language values and defining the location here. In this case, BASIC would have the location of the **string descriptor block** (not the string) ready to be picked up by a CALL 0A7FH.

Using VARPTR With Arrays

When VARPTR is used to find the location of an array, it has the same basic format as with strings. To find the location of array ZX, an integer array, you'd use:

```
100 A=VARPTR(ZX(0))
```

This finds the location of the first element in the array ZX. An integer array is made up of two-byte elements, a single-precision array of 4-byte elements, and a double-precision array of 8-byte elements.

For a one-dimensional array, the elements start with 0, 1, 2, etc. The 10th element of integer array ZX, for example, would be 20 bytes after VARPTR (ZX(0)).

For multiple-dimensioned arrays the format is more complicated, and we'll leave it up to you to research after this course (information on array formats is in the BASIC language manual for your Model I or III).

String arrays are not "contiguous" as are the other types of arrays. String array descriptor blocks are grouped together in one mass, however, as shown in Figure LESS25-2.

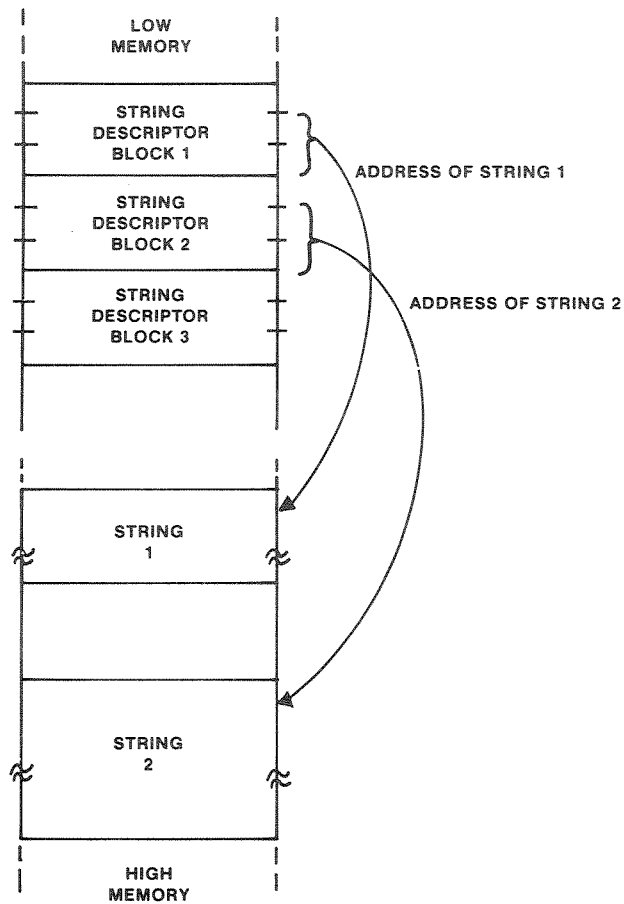


Figure LESS25-2. String Descriptors in String Array

To find the location of any string array, just use the index of the array variable as in

```
100 A=VARPTR(A$(5))
```

The address of the string descriptor block will be returned in A.

To show you how VARPTR works, enter the following program, or use the Lesson file:

25 VARPTR and Passing Multiple Arguments

```
100          ORG          0B800H
110 ;:.....
120 ;SUBROUTINE TO PRINT STRING IN REVERSE          *
130 ;*      ENTRY: STRING DESCRIPTOR BLOCK LOCATION          *
140 ;*      EXIT:  PRINT IN REVERSE ON SCREEN          *
150 ;:.....
160 PRTSTR      CALL      0A7FH          ;get location
170            PUSH      HL          ;move to IX
180            POP       IX
190            LD        E,(IX)        ;length of string
200            LD        L,(IX+1)      ;get ls byte of loc'n
210            LD        H,(IX+2)      ;get ms byte of loc'n
220            LD        B,E          ;in B for count
230            LD        D,0          ;length in DE
240            ADD       HL,DE         ;point to end+1
250            DEC       HL          ;end
260            LD        IX,3E10H      ;point to screen center
270            LD        A,B          ;get count
280            OR        A          ;test for 0
290            JR        Z,PRT090      ;go if "null" string
300 PRT010      LD        A,(HL)        ;get string character
310            LD        (IX),A        ;store on screen
320            DEC       HL          ;bump string pntr
330            INC       IX          ;bump screen pntr
340            DJNZ      PRT010        ;go if not done
350 PRT090      RET                   ;return to BASIC
360            END                   ;end
```

Assemble the code and convert the object to decimal values.

PRTSTR first calls the BASIC subroutine at 0A7FH to get the string descriptor block location. It assumes that something like A=VARPTR(A\$) has been done in the BASIC program and that the string location is waiting to be picked up.

The string descriptor block location in HL is then transferred to IX. E is loaded with the string length, which may be 0.

The next two loads load HL with the actual address of the string from the 2nd and 3rd bytes of the string descriptor block.

HL now contains the address of the string, whether it is in a BASIC statement or string variable storage.

The length in DE is now added to HL and the result is decremented by one to point to the last character in the string.

IX is loaded with the location of the center character position of video memory. As you recall, the video memory goes from 3C00H through 3FFFH, and the center minus a few character positions is at about 3E10H.

The string length in B is now tested for 0. If the length is 0, nothing will be printed on the screen, and a jump is made to the return point.

If the string length is not 0, the PRT010 loop prints the string in reverse on the screen, using the string pointer in HL, the screen pointer in IX, and the counter in B.

Try your hand at writing a BASIC program to interface with this assembly-language subroutine.

A BASIC program using this code is shown below. You can load it from cassette or disk with file "PRTSCN."

```

100 REM PRINT REVERSE STRING
110 CLS: CLEAR 1000
120 DATA 205,127,10,229,221,225,221,94,0,221,110,1,221,102,2
130 DATA 67,22,0,25,43,221,33,16,62,120,183,40,9,126,221,119,0
140 DATA 43,221,35,16,247,201
150 FOR I=47104 TO 47104+37
160 READ A
170 POKE I-65536,A
180 NEXT I
193 DEFUSRO=&HB800
200 SB=0
210 INPUT A$
220 CLS
230 SB=VARPTR(A$)
240 IF SB>32767 THEN SB=SB-65536
253 SR=USRO(SB)
260 GOTO 210

```

Even if your BASIC code does not look quite the same, the DATA values should be identical.

Run the program above or yours, and you should see any string that is input displayed across the screen in reverse. Hit BREAK to stop the program and reload ALT.

An important point about using BASIC programs: **VARPTR locations tend to move!** To use VARPTR properly, you must use it just before the USR call and not introduce "new" variables before the CALL. That's why we defined variable SB in line 190 instead of defining it with the VARPTR call. Any new variable may move all variables down, invalidating a VARPTR location.

Passing Multiple Arguments

We've seen how we can pass a single 16-bit argument or parameter to an assembly-language subroutine and how to pass one argument back. How can we pass several arguments? After all, even the subroutines we used earlier required more than one argument.

There are a number of ways to do this. If the arguments are small enough, then you can put them into H and L. If we had wanted to clear the video display from a given line and character position, then we might have had something like this:

```

;.....
;:CLEAR SCREEN FROM LINE X, CP Y
;:      ENTRY: (H)=LINE #, O — 15
;:      (L)=CP, O — 63
;.....

```

In this case the line values and character position values are each small enough to fit into a byte, and there's no reason why we can't pack them together.

Another way to pass multiple arguments is to use the 16-bit value passed from BASIC as a pointer to a "parameter block." The parameter block could be filled with POKEs from BASIC and could be in a predefined, protected area of RAM.

Suppose that we had the following parameters to pass to an assembly-language subroutine that searched a string array for a given string. We might have something like this

25 VARPTR and Passing Multiple Arguments

Location B000H=LS byte of first descriptor block
B001H=MS Byte of first descriptor block
B002H=LS byte of # elements in string array
B003H=MS byte of # elements in string array
B004H=LS byte of search string block address
B005H=MS byte of search string block address

We used an area of protected RAM to hold the parameter block. The first two bytes hold the address of the first descriptor block of the string array. The next two bytes hold the number of elements in the array. The last two bytes hold the address of the search string descriptor block. These values could all have been put into the parameter block by BASIC POKEs.

The search subroutine can output arguments in the same way. It can store the output parameters as follows:

Location B006H=LS byte of found string or -1
B007H=MS byte of found string or -1
B008H=element number if found

The BASIC program can then pick up the output parameters by PEEKs.

To Sum It All Up

To review what we've learned here:

- VARPTR is used in BASIC to find the location of a BASIC variable
- The location of any variable type may be found by VARPTR
- Each string in BASIC is defined by a "string descriptor block" that holds the string length in the first byte and the string location in the second and third bytes
- VARPTR locations may change between the VARPTR use and the USR call if previously undefined variables are used
- Multiple arguments may be "packed" into H and L if their number ranges are small enough
- Multiple arguments may also be passed via a parameter block in protected area of memory. This block is used by both BASIC and the assembly-language program as a "common" area
- POKEs and PEEKs can be used by BASIC to store and read values from the parameter block

For Further Study

BASIC PEEK (BASIC manual)
BASIC VARPTR (BASIC manual)
BASIC array formats (BASIC manual)

Lesson 26

Using ROM Subroutines

Load LESS26 from disk.

BASIC in the Model I or III is written in assembly language. There are certain subroutines, called "ROM subroutines" that are available to the user. In fact, there could be many different subroutines that you **might** use, but the ones we're going to discuss here are common subroutines whose definition will not change. Why? Primarily because they are documented in Radio Shack manuals as user-accessible entry points. It would be very difficult to document **all** possible subroutines. Later revisions to the BASIC interpreter might be very difficult if all subroutines had to remain fixed in location and input and output parameters.

There is a complete list of all ROM subroutines in the user manuals for Model I and III BASIC and in other Radio Shack manuals. We'll just be working with two ROM subroutines here. They are

1. Wait For Keyboard Character, Location 0049H
2. Display a Character, Location 0033H

Cautions On Using ROM Subroutines

One of the most important points that we can make about ROM subroutine use is that you must be aware of which registers the subroutine uses. This is true for **any** subroutine called, ROM subroutine or not, although we didn't stress this too greatly in previous lessons.

The Wait for Keyboard Character subroutine, for example, returns the character of the next key pressed on the keyboard in the A register. In doing so, it alters the DE register.

The Display Character subroutine displays the ASCII character in A, but in doing so, it also alters the DE register.

If you are using the DE register pair to hold a pointer value, or any other quantity, it will be destroyed after a return is made from either one of these subroutines. Prior to calling the subroutines, therefore, you must PUSH DE if it holds anything of value.

As a matter of fact, if you are unsure of which registers are used in a subroutine, there's no reason why you can't PUSH all of the registers, or at least all of the ones you're using. To save all registers only takes 6 instructions:

PUSH	AF	;save registers
PUSH	BC	
PUSH	DE	
PUSH	HL	
PUSH	IX	
PUSH	IY	
CALL	XXXXH	;call subroutine
POP	IY	;restore registers
POP	IX	
POP	HL	
POP	DE	
POP	BC	
POP	AF	

Using Display a Character and Wait for Character

Display a Character

This ROM subroutine is entered with a display character in the A register. The subroutine displays the character at the current screen location and then returns to the user program with DE altered. Doesn't

26 Using ROM Subroutines

sound like much, does it? In fact, though, you have all the power of the Display Driver at your disposal through this entry point. With the right character, you can clear the screen, tab, erase to end of line, move the cursor, or output graphics characters.

Using Display a Character is as simple as it looks. Put the character in A and CALL location 0033H.

Wait For a Character

This ROM subroutine is entered with no parameters and returns to the user program with an ASCII character representing the next keypress in the A register with DE destroyed. The character entered is not displayed on the screen. If no key is pressed Wait For a Character does not return!

Here again, this doesn't seem like a very powerful subroutine, but don't forget that implicit in this call are hundreds of bytes of instructions, including keyboard debouncing, "n" key rollover, and other processing.

With only these two ROM subroutines as a base, you can build a whole series of your own subroutines that can translate character strings, perform special display functions, do word processing, and other applications.

A Simple Text Editor

To show you how these two subroutines can be used to build on, we've written a simple text editor. This program will utilize the Wait For Character and Display Character subroutines to implement a "stand-alone" text editor that will allow you to enter text and store it on the screen. The screen cursor is controlled by the up arrow, down arrow, right arrow, and left arrow keys. You may move the cursor anywhere on the screen that you wish to initiate new text or to overwrite old.

Enter the program below, or use the Lesson file:

```
110 ;.....
120 ;• MINI TEXT EDITOR *
130 ;.....
140 MINITE      CALL      HOME      ;home cursor
150            LD         A,1FH      ;erase to end of disp
160            CALL      33H        ;output
170            CALL      HOME      ;home cursor
180            LD         A,OEH      ;cursor on
190            CALL      33H        ;output
200 TXT010      CALL      49H        ;input character
210            LD         HL,FTAB     ;function key table
220            LD         BC,4        ;size of table
230            CPIR              ;search for key
240            JR         Z,TXT020    ;go if found
250 TXT015      CALL      33H        ;not fnd, output char
260            JR         TXT010     ;go for next char
270 TXT020      LD         BC,FTABP1  ;start+1
280            OR         A          ;clear carry
290            SBC         HL,BC      ;find index
300            PUSH      HL          ;index to IX
310            POP       IX
320            ADD         IX,IX      ;index*2
330            LD         BC,BTAB     ;branch table start
340            ADD         IX,BC      ;point to BR address
350            LD         L,(IX)      ;get BR address
360            LD         H,(IX+1)    ;
370            JP         (HL)        ;branch out
380 FTAB        DEFB         5BH     ;up arrow
```

390 FTABP1	DEFB	0AH	;down arrow
400	DEFB	09H	;right arrow
410	DEFB	08H	;left arrow
420 BTAB	DEFW	UPARR	;up arrow proc
430	DEFW	DWNARR	;down arrow proc
440	DEFW	RGTARR	;right arrow proc
450	DEFW	LFTARR	;left arrow proc
460 UPARR	LD	A,1BH	;move cursor up
470	JR	TXTO15	;output
480 DWNARR	LD	A,1AH	;move cursor down
490	JR	TXTO15	;output
500 RGTARR	LD	A,19H	;advance cursor
510	JR	TXTO15	;output
520 LFTARR	LD	A,18H	;backspace
530	JR	TXTO15	;output
540 HOME	LD	A,1CH	;move cursor upper left
550	CALL	33H	;output
560	RET		;return
570	END		;end

For this program only we have disabled the normal controls of ALT so that you can execute the program without ALT interference. Assemble the program and get an error-free assembly. Now execute the program by transferring control to the starting address (check the listing) by

ZXG XXXX

where XXXX is the starting address

Once this is done, by the way, you'll have to reload ALT and the Lesson File.

You'll see the program clear the screen and position the cursor in the upper left-hand corner of the display, the HOME position. You can now enter upper-case text and move the cursor around with the arrows.

The Wait For Character subroutine at 33H returns a code corresponding to the key pressed. Usually this is an ASCII code corresponding to an alphabetic character, numeric character, or special character, such as "#."

The subroutine, however, also returns codes for special keys, such as the arrow keys, ENTER, and others. You'll find a complete list in the back of your BASIC manual. The ones we'll be considering here are the codes for up arrow (5BH), down arrow (0AH), right arrow (09H), and left arrow (08H).

On the output, or display side, the Display Character subroutine at 49H normally displays alphabetic, numeric, or special characters, but also uses special codes to perform certain functions. Outputting a 1CH, for example, moves the cursor to the upper-left hand corner of the screen, the so-called HOME position.

The 1BH code moves the cursor up; at the top boundary, the cursor appears at the bottom of the screen. The 1AH code moves the cursor down; at the bottom boundary, the cursor appears at the top of the screen. The 19H code advances the cursor to the next character position. The 18H code "backspaces" the cursor to the left without deleting a character.

The cursor itself is turned on by a 0EH character.

In general, most of the special character input and output codes are below 20H, the first "text" character, a space.

Let's take a more detailed look at the program:

There's one subroutine called HOME. This simply outputs a HOME code by calling the Display Character subroutine in ROM.

26 Using ROM Subroutines

HOME is called to initialize the display. Next, a 1FH code is output to Display Character. This is a special code to “erase from the current cursor position to the end of the display.” The current cursor position is upper left, which was automatically established in the display driver when the HOME code was output.

Let’s stress here that the major part of the display effort — computing addresses, blinking the cursor, maintaining the cursor position, “automatic” typing when the key is held down — is all done in the ROM display driver, accessed by the CALL to 33H.

After the display is cleared, a CALL to HOME returns the cursor to the upper left position.

TXT010 starts the main loop of the program.

The next keypress is input from the Wait for Character subroutine. A scan is then made through the FTAB table to see if the input character matches any code in the table. There are 4 codes in the table, corresponding to the arrow key codes. At the end of the CPIR, the HL register points to the character+1 if the character has been found (Z).

If the character has not been found, then the character is a “normal” text character. It’s output to the display by CALLing 33H at TXT015. Note that up to this point, no character has been output; reading in the character does not automatically display it. After the output, a loop is made back to TXT010 to input and display the next character.

If the character is found in the FTAB, then the start address of FTAB+1 is subtracted from the contents of HL. HL now contains an index of 0, 1, 2, or 3. This index value is doubled and added to the starting address of BTAB, a “branch table.” At the end of the add, IX points to a branch address corresponding to the processing for the special key. HL is then loaded with the branch address from the table, and a JP (HL) branches out to one of the four processing routines.

Each of the 4 processing routines simply outputs the Display Character code corresponding to the special function and then returns to TXT010 for the next input character.

Using ROM Subroutines for Your Own Code

The simple program above shows you how you can take advantage of some of the existing ROM subroutines to eliminate a lot of tedious coding. Look for other examples of keyboard input processing, display output, line printer output, cassette operations, and disk operations in your BASIC and other manuals.

To Sum It All Up

To review what we’ve learned in this lesson:

- There are a number of documented ROM subroutines that can be used to eliminate your own assembly-language coding for keyboard input, display operations, and others
- When using these subroutines, or any subroutines, you must be aware of which registers may be destroyed by the action of the subroutine; save these registers by PUSHes before the subroutine call
- Display Character subroutine outputs one character to the video display and uses the full logic of the BASIC display driver software
- Wait For Character inputs the next keypress from the keyboard input driver
- Special character codes exist for both keyboard input and display output; they are separate from the normal input and output of ASCII text and are usually in the 00H through 1FH range

For Further Study

Character codes for input and output (BASIC manual)
ROM subroutines (BASIC and other manuals)

Lesson 27

Where Do You Go From Here?

There is no Lesson File for this lesson.

In the past lessons, we've given you a description of the instruction set and addressing modes of the Z-80, provided some instruction on using a typical editor/assembler for assembly-language programs, given you techniques for interfacing to BASIC, and described some BASIC ROM subroutines that simplify coding. How do you put all of this information together to write your own assembly-language programs?

In this final lesson, we'll look at a plan for doing this. The plan consists of 5 parts — design, flowcharting, coding, debugging, and documentation.

Program Design

This is the first stage of any program, whether it is BASIC programming or assembly-language programming. Program design for large programs in many cases consists of writing the “design specification” for the program before anything else is done.

The design spec is a detailed manual outlining what the program will do and in general how it will go about it. All screen formats, record formats, menus, and commands are listed and detailed.

Of course, for small programs you don't really need this design spec. If you were implementing a bubble sort, for example, you know that the sort has to put all of the data items of a table in sequence, and not too much more can be said about it.

This design phase of a program is still critical, however, even though you don't write a design spec. You should spend some time thinking about the general “dimensions” of your problem. In the case of a bubble sort, for example, you might ask yourself how many entries will be sorted, what the maximum number of entries are, whether the number of entries can be held in 8 bits or 16, where the table will be located, how large the size of entries will be, and so forth.

If you don't give the problem some thought, you may find yourself in the middle of a program that simply won't work because it can't!

Program Flowcharting

The next step in any type of programming is flowcharting. We've used a few flowcharts in these lessons, so you're somewhat familiar with the symbols.

For a recap, the symbols are shown in Figure LESS27-1.

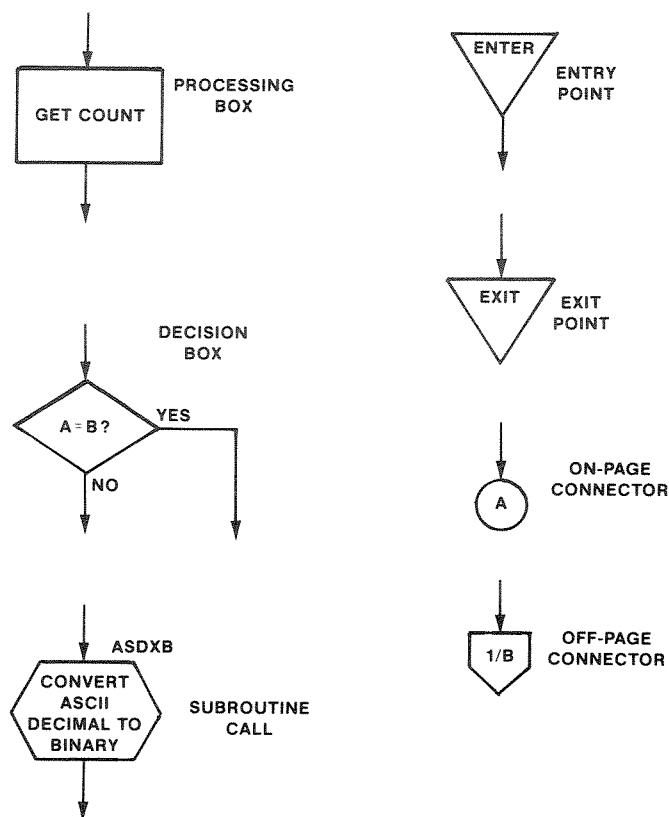


Figure LESS27-1. Flowcharting Symbols

The rectangular box is a “processing box.” Any type of general processing, such as “LOAD NUM WITH 0,” “GET NEXT CHARACTER,” or “BUMP POINTER BY 1” is put into the box to describe the processing action.

The diamond is a decision box. The decision box would be equated to conditional jumps in assembly language. Two or more exit points might be used from the decision box. You might have something like “A<B?” with one branch labeled “YES” and the other “NO,” or you might have “MORE ENTRIES?” with another set of “YES” and “NO” exits.

I’ve used the six-sided symbol for subroutine processing, although this is not always standard. It’s customary to put the name of the subroutine at the upper right of the box. The description within the subroutine box might be something like “SCAN TABLE FOR LEAST ENTRY” or “CONVERT TO BINARY.”

The triangles are exit and entry points from a subroutine or other code. Typical descriptive text would be “ENTER” or “ENTER FROM COMMAND INTERPRETER” in the entry points and “RETURN” or “RETURN TO MAIN” in the exit points.

The circles are “on-page connectors.” They are usually labeled with alphabetical letters. Two circles on each page will have the same designator, showing how the program flows without having to draw confusing lines.

The spade-shaped symbols are “off-page connectors,” which connect a program point to a continuing point on another page. The off-page connectors are usually labeled with page numbers.

Typically, labels at the top left of each flowchart symbol represent the label that will be used in the assembly listing.

The above flowchart symbols are suggestions only, although the rectangle and diamond are standard symbols.

Flowcharts usually flow down and to the right.

When flowcharts are done before a program is coded, it makes it very easy for a programmer to see what is happening without having to wade through dozens of instructions.

Do you need to flowchart? Not for simple programs. For programs that involve dozens of instructions, a quick rough flowchart helps clarify what you're going to do, however. For larger programs, flowcharts are a must. They help you plan the code and provide an indication of how you did things when you come back to the program six months later!

Some programmers use notes as an alternative to flowcharting. Use flowcharts first for larger programs, and then find a technique that seems to work for you.

If you are flowcharting your program, try to divide a large program up into subroutines and other "modules" as much as possible, rather than having one huge set of continuous code.

A module would in many cases be a subroutine with the entry conditions and exit conditions very well defined, as we've done in some of the subroutines in this text.

In other cases, a module would simply be a functional processing block that performed a specific function, such as doing the "insert" function of a word-processing program.

Program Coding

After some thought about program design and some flowcharting for larger programs, you're ready to code. You may find that your flowcharts are detailed enough so that you can just sit down and enter instructions directly into the editor/assembler. Chances are, though, you'll have to code your program first using paper and pencil.

Once you've coded the program on paper and given it a cursory check, you're ready to enter it into the editor/assembler.

A few tips about program structure:

- Programs usually flow through from beginning to end.
- Try to utilize as many subroutines and modules as possible.
- Use labels for subroutine and module entry points that correspond to the function — "RDCHAR" for "Read Character," for example. Labels after this entry point may be defined by the first 3 letters of the function and then 3 digits in ascending order. You might have the labels "RDC010," "RDC030," and "RDC100" as three labels in the RDCHAR subroutine, for example.
- Use as many comments as possible.
- Format your programs with "pretty printing." Bracket subroutines with asterisks or other symbols, and try to create listings that are easy to read.
- Use labels that correspond to the flowchart location points. You might want to go back and add the labels to the flowchart after you've coded them.
- Labels are not necessary unless they define jump locations or subroutine locations. They just take up space in the assembler symbol table otherwise.

When you assemble the program, don't be disappointed if you get dozens of errors. This happens to every programmer at different times. Typically, it might take 3 passes to get rid of all errors.

Once you have a "clean program" without errors, you're ready to do program debugging.

Program Debugging

The first step of debugging is called "desk checking." Sit down at your desk and carefully go over the program listing. You may even want to "play computer" and pencil in the registers and stack, and then

27 Where Do You Go From Here

follow the program flow. Chances are you'll uncover some errors that will necessitate reassembly. Again, don't be dismayed, few programmers can write programs that work the first time.

After you've thoroughly desk checked the code and made any necessary changes and reassemblies, you're ready for "on-line" debugging.

If you're using ALT, there are some built-in debugging tools that will help you:

- The ZB command lets you set breakpoints to stop the program at any location, once that location is reached.
- The ZT command lets you trace any memory area either while running the program or afterward
- The ZS command and ZX command let you execute the program at slow speed and observe the registers and memory trace area.
- The ZR command and ZM command let you change registers or memory to "dummy up" data or establish test cases.

If you're using the Series I Editor/Assembler, you'll need a DEBUG package, available on either cassette or disk. The DEBUG package will give you most of the ALT features and some additional ones that will help in debugging.

There is no set procedure for debugging, but here are some general guidelines:

- Use breakpoints to narrow down where an error is occurring. Breakpoint at one location and see if the location is reached. If not, go to the halfway point and breakpoint there, and so forth, until you find the spot where the error occurred.
- ALT will give an error message if you attempt to jump out of the program area. It gives you full control over program execution.
- If you are not using ALT, always reload the program if it "blows up," instead of restarting the DEBUG package. Incorrect program operation might have destroyed parts of your program, and this will create further errors or misleading information.
- Be aware of which instructions affect the Flags and which ones do not. INCs and DEC of register pairs, for example, NEVER affect the Flags.
- Make certain that you handle the stack properly. You should have a PUSH for every POP and an RET for every CALL.

Program Documentation

Once you have a final version of a program, sit down and write a brief summary of how it works. Include "internal" tables and data structures and a description of program variables, if the program is large. If you have a simple subroutine interfacing to BASIC that is commented on the listing, this step is not necessary.

To Sum It All Up

To review what we've learned in this lesson:

- Program development consists of program design, flowcharting, coding, debugging, and documentation
- The design phase consists of a program specification, or at least some serious thought about what the program is to accomplish and how it will go about it

- Somewhat standardized flowchart symbols are used to layout the program flow in “schematic” form
- Coding should first be done on paper and then entered into the Editor/Assembler
- Debugging consists of desk checking and actual “on-line” debugging using a DEBUG package to trace down errors
- Final documentation may be written for larger programs to describe program operation

For Further Study

Model I/III Series I Editor/Assembler

“TRS-80 Assembly-Language Programming,” Radio Shack 26-2006, by William Barden, Jr.

”More TRS-80 Assembly-Language Programming,” Radio Shack 62-2075, by William Barden, Jr.



Appendix I. ALT Commands

Command	Format	Description
A	A	Assembles current source code
	A WE	Assembles with "wait on error"
	A LP	Assembles with line printer output
D	DLLL	Delete source line LLLL
	D#	Delete starting line
	D*	Delete ending line
	DLLL:MMM	Delete lines LLL through MMM (* , #, or period may be used)
H	HLLL:MMM	Hardcopy lines LLL through MMM to line printer (# , * , or period may be used)
I	ILLL,II	If no source lines, start new source from LLLL with increment II
		If source lines, insert between lines with line number LLLL and increment II
L	L	Load next file from cassette or file "NONAME" from disk
	L NAME	Load file "NAME" from cassette or disk
N	NLLL,II	Renumber source lines starting with line number LLL and increment II
P	P	Display next 5 lines
	P#	Display from starting line
	P*	Display ending line
	PLLL	Display from line LLL
Q	Q	Quit. Go back to BASIC or TRSDOS
W	W	Write file on cassette or disk with name "NONAME"
	W NAME	Write file on disk or cassette with name "NAME"
ZB	ZB MMMM	Breakpoint instruction at memory location hex MMMM
ZM	ZM MMMM	Modify memory location hex MMMM
ZR	ZR R=	Modify register R (A, F, B, C, D, E, H, L, AF, BC, DE, HL, IX, IY, SP, PC)
ZS	ZS NNNN	Set speed from 0 (slowest) to 9999, NNNN in decimal
ZT	ZT MMMM	Trace memory location hex MMMM
	ZTT MMMM	Trace memory location hex MMMM in ASCII
	ZT	Switch to hexadecimal trace
	ZTT	Switch to ASCII trace
	ZT+	Trace next 32 memory locations
	ZT-	Trace last 32 memory locations
	ZX	Execute from start of object file
ZX	ZX MMMM	Execute from hex address MMMM
	ZXG MMMM	Execute from hex address MMMM without ALT control
	ZXS	Execute, single step by keypress
ZZ	ZZ MMMM	Zap (delete) breakpoint at instruction at memory location hex MMMM



Appendix II. ALT Assembler Pseudo-Ops

<u>Pseudo Op</u>	<u>Format</u>			<u>Description</u>
DEFB	(LABEL)	DEFB	NN NNH	Generates one byte in decimal Generates one byte in hex
DEFM	(LABEL)	DEFM	'string'	Generates a string of chars Only first 4 characters printed
DEFW	(LABEL)	DEFW	NNNN NNNNH	Generates two bytes in decimal Generates two bytes in hex
DEFS	(LABEL)	DEFS	NNNN NNNNH	Reserves NNNN (decimal) bytes Reserves NNNN (hex) bytes
END	(LABEL)	END		Terminates interpreter
ORG	(LABEL)	ORG	NNNN	Establishes Origin of next set of code. Either NNNN (decimal) or NNNNH (hex) may be used.

(LABEL)=optional label



Appendix III. Binary/Decimal/Hexadecimal Conversions

HEX	BINARY	DECIMAL
00	00000000	0
01	00000001	1
02	00000010	2
03	00000011	3
04	00000100	4
05	00000101	5
06	00000110	6
07	00000111	7
08	00001000	8
09	00001001	9
0A	00001010	10
0B	00001011	11
0C	00001100	12
0D	00001101	13
0E	00001110	14
0F	00001111	15
10	00010000	16
11	00010001	17
12	00010010	18
13	00010011	19
14	00010100	20
15	00010101	21
16	00010110	22
17	00010111	23
18	00011000	24
19	00011001	25
1A	00011010	26
1B	00011011	27
1C	00011100	28
1D	00011101	29
1E	00011110	30
1F	00011111	31
20	00100000	32
21	00100001	33
22	00100010	34
23	00100011	35
24	00100100	36
25	00100101	37
26	00100110	38
27	00100111	39
28	00101000	40
29	00101001	41
2A	00101010	42
2B	00101011	43
2C	00101100	44
2D	00101101	45
2E	00101110	46
2F	00101111	47
30	00110000	48
31	00110001	49
32	00110010	50
33	00110011	51

APPENDIX III Binary/Decimal/Hexadecimal Conversions

34	00110100	52
35	00110101	53
36	00110110	54
37	00110111	55
38	00111000	56
39	00111001	57
3A	00111010	58
3B	00111011	59
3C	00111100	60
3D	00111101	61
3E	00111110	62
3F	00111111	63
40	01000000	64
41	01000001	65
42	01000010	66
43	01000011	67
44	01000100	68
45	01000101	69
46	01000110	70
47	01000111	71
48	01001000	72
49	01001001	73
4A	01001010	74
4B	01001011	75
4C	01001100	76
4D	01001101	77
4E	01001110	78
4F	01001111	79
50	01010000	80
51	01010001	81
52	01010010	82
53	01010011	83
54	01010100	84
55	01010101	85
56	01010110	86
57	01010111	87
58	01011000	88
59	01011001	89
5A	01011010	90
5B	01011011	91
5C	01011100	92
5D	01011101	93
5E	01011110	94
5F	01011111	95
60	01100000	96
61	01100001	97
62	01100010	98
63	01100011	99
64	01100100	100
65	01100101	101
66	01100110	102
67	01100111	103
68	01101000	104
69	01101001	105

6A	01101010	106
6B	01101011	107
6C	01101100	108
6D	01101101	109
6E	01101110	110
6F	01101111	111
70	01110000	112
71	01110001	113
72	01110010	114
73	01110011	115
74	01110100	116
75	01110101	117
76	01110110	118
77	01110111	119
78	01111000	120
79	01111001	121
7A	01111010	122
7B	01111011	123
7C	01111100	124
7D	01111101	125
7E	01111110	126
7F	01111111	127
80	10000000	128
81	10000001	129
82	10000010	130
83	10000011	131
84	10000100	132
85	10000101	133
86	10000110	134
87	10000111	135
88	10001000	136
89	10001001	137
8A	10001010	138
8B	10001011	139
8C	10001100	140
8D	10001101	141
8E	10001110	142
8F	10001111	143
90	10010000	144
91	10010001	145
92	10010010	146
93	10010011	147
94	10010100	148
95	10010101	149
96	10010110	150
97	10010111	151
98	10011000	152
99	10011001	153
9A	10011010	154
9B	10011011	155
9C	10011100	156
9D	10011101	157
9E	10011110	158
9F	10011111	159

APPENDIX III Binary/Decimal/Hexadecimal Conversions

<u>HEX</u>	<u>BINARY</u>	<u>DECIMAL</u>
A0	10100000	160
A1	10100001	161
A2	10100010	162
A3	10100011	163
A4	10100100	164
A5	10100101	165
A6	10100110	166
A7	10100111	167
A8	10101000	168
A9	10101001	169
AA	10101010	170
AB	10101011	171
AC	10101100	172
AD	10101101	173
AE	10101110	174
AF	10101111	175
B0	10110000	176
B1	10110001	177
B2	10110010	178
B3	10110011	179
B4	10110100	180
B5	10110101	181
B6	10110110	182
B7	10110111	183
B8	10111000	184
B9	10111001	185
BA	10111010	186
BB	10111011	187
BC	10111100	188
BD	10111101	189
BE	10111110	190
BF	10111111	191
C0	11000000	192
C1	11000001	193
C2	11000010	194
C3	11000011	195
C4	11000100	196
C5	11000101	197
C6	11000110	198
C7	11000111	199
C8	11001000	200
C9	11001001	201
CA	11001010	202
CB	11001011	203
CC	11001100	204
CD	11001101	205
CE	11001110	206
CF	11001111	207
D0	11010000	208
D1	11010001	209
D2	11010010	210
D3	11010011	211
D4	11010100	212

D5	11010101	213
D6	11010110	214
D7	11010111	215
D8	11011000	216
D9	11011001	217
DA	11011010	218
DB	11011011	219
DC	11011100	220
DD	11011101	221
DE	11011110	222
DF	11011111	223
E0	11100000	224
E1	11100001	225
E2	11100010	226
E3	11100011	227
E4	11100100	228
E5	11100101	229
E6	11100110	230
E7	11100111	231
E8	11101000	232
E9	11101001	233
EA	11101010	234
EB	11101011	235
EC	11101100	236
ED	11101101	237
EE	11101110	238
EF	11101111	239
F0	11110000	240
F1	11110001	241
F2	11110010	242
F3	11110011	243
F4	11110100	244
F5	11110101	245
F6	11110110	246
F7	11110111	247
F8	11111000	248
F9	11111001	249
FA	11111010	250
FB	11111011	251
FC	11111100	252
FD	11111101	253
FE	11111110	254
FF	11111111	255



Appendix IV. Conversion Techniques

To Convert From Binary or Hexadecimal to Decimal:

1. If number is 8 bits or less, use Appendix III.
2. If number is greater than 8 bits, use this method:
 - A. Divide into two bytes (add zeroes to left if necessary)
 - B. Convert first (most significant) by Appendix III.
 - C. Multiply decimal equivalent of first by 256.
 - D. Convert second (least significant) by Appendix III.
 - E. Add the result of C and D together to find the decimal number.
 - F. Example: Convert 0AA88H to decimal.
 - a. First byte is AA, second is 88.
 - b. From Appendix III, first is 170 in decimal.
 - c. The value 170 multiplied by 256 is 43,520.
 - d. Second byte of 88H is 136 decimal from Appendix III.
 - e. $43,520 + 136$ is 43,656 decimal = 0AA88H.

To Convert From Decimal to Binary or Hexadecimal:

1. If number is 255 or less, use Appendix III.
2. If number is greater than 255, use this method:
 - A. Divide by 256 to get an integer result and a remainder.
 - B. Convert integer result to a hexadecimal or binary number by Appendix III.
 - C. Convert remainder to a hexadecimal or binary number by Appendix III.
 - D. Write down the hex number from B followed by hex number from C; you should have four hex digits or 16 binary digits. The result is the number in hex or binary.
 - E. Example: Convert 60000 to hexadecimal.
 - a. $60000 / 256 = 234$, remainder of 96.
 - b. From Appendix III, integer 234 is EA in hexadecimal.
 - c. From Appendix III, remainder 96 is 60 in hexadecimal.
 - d. EA followed by 60 is 0EA60H = 60000 decimal.



APPENDIX V

Z-80 Instruction Set

A Register Operations

Complement PL
Decimal DAA
Negate NEG

Adding/Subtracting Two 8-Bit Numbers

A and Another Register

ADC A,r SBC A,r
ADD A,r SUB A,r

A and Immediate Operand

ADC A,n SBC A,n
ADD A,n SUB A,n

A and Memory Operand

ADC A,(HL) ADD A,(HL) SBC (HL) SUB (HL)
ADC A,(IX+d) ADD A,(IX+d) SBC (IX+d) SUB (IX+d)
ADC A,(IY+d) ADD A,(IY+d) SBC (IY+d) SUB (IY+d)

Adding/Subtracting Two 16-Bit Numbers

HL and Another Register Pair

ADC HL,ss ADD HL,ss SBC HL,ss

IX and Another Register Pair

ADD IX,pp ADD IY,rr

Bit Instructions

Test Bit

Register BIT b,r
Memory BIT b,(HL) BIT b,(IX+d) BIT b,(IY+d)

Reset Bit

Register RES b,r
Memory RES b,(HL) RES b,(IX+d) RES b,(IY+d)

Set Bit

Register SET b,r
Memory SET b,(HL) SET b,(IX+d) SET b,(IY+d)

Carry Flag

Complement CCF
Set SCF

Compare Two 8-Bit Operands

A and Another Register CP r
A and Immediate Operand CP n
A and Memory Operand
CP (HL) CP (IX+d) CP (Y+d)
Block Compare
CPD,CPDR,CPI,CPIR

Decrements and Increments

Single Register

DEC r INC r DEC IX DEC IY INC

Register Pair

DEC ss INC ss DEC IX DEC IY INC IX DEC IY

Memory

DEC HL DEC (IX+d) DEC (IY+d)

APPENDIX V Z-80 Instruction Set

Exchanges

DE and HL EX DE,HL
Top of Stack
EX (SP),HL EX (SP),IX EX (SP),IY

Input/Output

I/O To/From A and Port
IN A,(n) OUT (n),A
I/O To/From Register and Port
IN r,(C) OUT (C),r
Block
IND,INDR,INR,INIR,OTDR,OTIR,OUTD,OUTI

Interrupts

Disable DI
Enable EI
Interrupt Mode
IM 0 IM 1 IM 2
Return From Interrupt
RETI RETN

Jumps

Unconditional
JP (HL) JP (IX) JP (IY) JP (nn) JR e
Conditional
JP cc,nn JR C,e JR NZ,e Z,e
Special Conditional
DJNZ e

Loads

A Load Memory Operand
LD A,(BC) LD A,(DE) LD A,(nn)
A and Other Registers
LD A,I LD A,R LD I,A LD R,A
Between Registers, 8-Bit
LD r,r'
Immediate 8-Bit
LD r,n
Immediate 16-Bit
LD dd,nn LD IX,nn LD IY,nn
Register Pairs From Other Register Pairs
LD SP,HL LD SP,IX LD SP,IY
From Memory, 8 Bits
LD r,(HL) LD r,(IX+d) LD r,(IY+d)
From Memory, 16 Bits
LD HL,(nn) LD IX,(nn) LD IY,(nn) LD dd,(nn)
Block
LDD,LDDR,LDI,LDIR

Logical Operations 8 Bits With A

A and Another Register

AND r OR r XOR r

A and Immediate Operand

AND n OR n XOR n

A and Memory Operand

AND (HL) OR (HL) XOR (HL)

AND (IX+d) OR (IX+d) XOR (IX+d)

AND (IY+d) OR (IY+d) XOR (IY+d)

Miscellaneous

Halt HALT

No Operation NOP

Prime/Non-Prime

Switch AF

EX AF,AF'

Switch Others

EXX

Shifts

Circular (Rotate)

A Only RLA, RLCA, RRA, RRCA

All Registers Rl r RLC r RR r RRC r

Memory

RL (HL) RLC (HL) RR (HL) RRC (HL)

RL (IX+d) RLC (IX+d) RR (IX+d) RRC (IX+d)

RL (IY+d) RLC (IY+d) RR (IY+d) RRC (IY+d)

Logical

Registers SRL r

Memory SRL (HL) SRL (IX+d) SRL (IY+d)

Arithmetic

Registers SLA r SRA r

Memory

SLA (HL) SRA (HL)

SLA (IX+d) SRA (IX+d)

SLA (IY+d) SRA (IY+d)

Binary-Coded-Decimal

RLD RRD

Stack Operations

PUSH IX PUSH IY PUSH qq POP IX POP IY POP qq

Stores

Of A Only

LD (BC),A LD (DE),A LD (nn),A

All Registers

LD (HL),r LD (IX+d),r LD (IY+d),r

Immediate Data

LD (HL),n LD (IX+d),n LD (IY+d),n

16-Bit Registers

LD (nn),dd LD (nn),IX LD (nn),IY LD (nn),HL

APPENDIX V Z-80 Instruction Set

Subroutine Action

Conditional CALLs CALL cc,nn

Unconditional CALLs CALL nn

Conditional Return RET cc

Unconditional Return RET

Special CALL RST p

Mnemonic	Format	Description	S	Z	P/V	C
ADC HL,ss	11101101 01ss1010	HL+ss+CY to HL	•	•	•	•
ADC A,r	10001 r	A+r+Cy to A	•	•	•	•
ADC A,n	11001110 n	A+n+Cy to A	•	•	•	•
ADC A,(HL)	10001110	A+(HL)+Cy to A	•	•	•	•
ADC A,(IX+d)	11011101 10001110 d	A+(IX+d)+CY to A	•	•	•	•
ADC A,(IY+d)	11111101 10001110 d	A+(IY+d)+CY to A	•	•	•	•
ADD A,n	11000110 n	A+n to A	•	•	•	•
ADD a,r	10000 r	A+r to A	•	•	•	•
ADD A,(HL)	10000110	A+(HL) to A	•	•	•	•
ADD A,(IX+d)	11011101 10000110 d	A+(IX+d) to A	•	•	•	•
ADD A,(IY+d)	11111101 10000110 d	A+(IY+d) to A	•	•	•	•
ADD HL,ss	00ss1001	HL+ss to HL				•
ADD IX,pp	11011101 00pp1001	IX+pp to IX				•
ADD IY,rr	11111101 00rr1001	IY+rr to IY				•
AND r	10100 r	A AND r to A	•	•	•	0
AND n	11100110 n	A AND n to A	•	•	•	0
AND (HL)	10100110	A AND (HL) to A	•	•	•	0
AND (IX+d)	11011101 10100110 d	A AND (IX+d) to A	•	•	•	0
AND (IY+d)	11111101 10100110 d	A AND (IY+d) to A	•	•	•	0
BIT b,r	11001011 01 b r	Test bit b of r	•	•	•	
BIT b,(HL)	11001011 01 b 110	Test bit b of (HL)	•	•	•	
BIT b,(IX+d)	11011101 11001011 d 01 b 110	Test bit b of (IX+d)	•	•	•	
BIT b,(IY+d)	11111101 11001011 d 01 b 110	Test bit b of (IY+d)	•	•	•	
CALL cc,nn	11 c 100 n n	CALL subroutine at nn if cc				
CALL nn	11001101 n n	Unconditionally CALL nn				
CCF	00111111	Complement carry flag				•
CP r	10111 r	Compare A:r	•	•	•	•
CP n	11111110 n	Compare A:n	•	•	•	•
CP (HL)	10111110	Compare A:(HL)	•	•	•	•
CP (IX+d)	11011101 10111110 d	Compare A:(IX+d)	•	•	•	•
CP (IY+d)	11111101 10111110 d	Compare A:(IY+d)	•	•	•	•
CPD	11101101 10101001	Block Compare, no repeat	•	•	•	
CPDR	11101101 10111001	Block Compare, repeat	•	•	•	
CPI	11101101 10100001	Block Compare, no repeat	•	•	•	
CPIR	11101101 10110001	Block Compare, repeat	•	•	•	
CPL	00101111	Complement A (1's comple)				
DAA	00100111	Decimal Adjust A	•	•	•	
DEC r	00 r 101	Decrement r by one	•	•	•	

Mnemonic	Format			Description	S	Z	P/V	C
DEC (HL)	00110101			Decrement (HL) by one	•	•	•	
DEC (IX+d)	11011101	00110101	d	Decrement (IX+d) by one	•	•	•	
DEC (IY+d)	11111101	00110101	d	Decrement (IY+d) by one	•	•	•	
DEC IX	11011101	00101011		Decrement IX by one				
DEC IY	11111101	00101011		Decrement IY by one				
DEC ss	00ss1011			Decrement register pair				
DI	11110011			Disable interrupts				
DJNZ e	00010000	e-2		Decrement B and JR if B≠0				
EI	11111011			Enable interrupts				
EX (SP),HL	11100011			Exchange (SP) and HL				
EX (SP),IX	11011101	11100011		Exchange (SP) and IX				
EX (SP),IY	11111101	11100011		Exchange (SP) and IY				
EX AF,AF'	00001000			Set prime AF active				
EX DE,HL	11101011			Exchange DE and HL				
EXX	11011001			Set prime B-L active				
HALT	01110110			Halt				
IM 0	11101101	01000110		Set interrupt mode 0				
IM 1	11101101	01010110		Set interrupt mode 1				
IM 2	11101101	01011110		Set interrupt mode 2				
IN A,(n)	11011011	n		Load A with input from n				
IN r,(C)	11101101	01 r 000		Load r with input from (C)	•	•	•	
INC r	00 r 100			Increment r by one	•	•	•	
INC (HL)	00110100			Increment (HL) by one	•	•	•	
INC (IX+d)	11011101	00110100	d	Increment (IX+d) by one	•	•	•	
INC (IY+d)	11111101	00110100	d	Increment (IY+d) by one	•	•	•	
INC IX	11011101	00100011		Increment IX by one				
INC IY	11111101	00100011		Increment IY by one				
INC ss	00ss0011			Increment register pair				
IND	11101101	10101010		Block I/O input from (C)	•	•	•	
INDR	11101101	10111010		Block I/O input, repeat	•	•	•	
INI	11101101	10100010		Block I/O input from (C)	•	•	•	
INIR	11101101	10110010		Block I/O input, repeat	•	•	•	
JP (HL)	11101001			Unconditional jump to (HL)				
JP (IX)	11011101	11101001		Unconditional jump to (IX)				
JP (IY)	11111101	11101001		Unconditional jump to (IY)				
JP cc,nn	11 c 010	n	n	Jump to nn if cc				
JP nn	11000011	n	n	Unconditional jump to nn				
JR C,e	00111000	e-2		Jump relative if carry				
JR e	00011000	e-2		Unconditional jump relative				
JR NC,e	00110000	e-2		Jump relative if no carry				
JR NZ,e	00100000	e-2		Jump relative if non-zero				
JR Z,e	00101000	e-2		Jump relative if zero				
LD A,(BC)	00001010			Load A with (BC)				

APPENDIX V Z-80 Instruction Set

Mnemonic	Format	Description	S	Z	P/V	C
LD A,(DE)	00011010	Load A with (DE)				
LD A,I	11101101 01010111	Load A with I	•	•	•	
LD A,(nn)	00111010 n n	Load A with location nn				
LD A,R	11101101 01011111	Load A with R	•	•	•	
LD (BC),A	00000010	Store A to (BC)				
LD (DE),A	00010010	Store A to (DE)				
LD (HL),n	00110110 n	Store n to (HL)				
LD dd,nn	00dd0001 n n	Load register pair with nn				
LD dd,(nn)	11101101 01dd1011 n n	Load register pair with location nn				
LD HL,(nn)	00101010 n n	Load HL with location nn				
LD (HL),r	01110 r	Store r to (HL)				
LD I,A	11101011 01000111	Load I with A				
LD IX,(nn)	11011101 00101010 n n	Load IX with nn				
LD IX,nn	11011101 00100001 n n	Load IX with location nn				
LD (IX+d),n	11011101 00110110 d n	Store n to (IX+d)				
LD (IX+d),r	11011101 01110 r d	Store r to (IX+d)				
LD IY,nn	11111101 00100001 n n	Load IY with nn				
LD IY,(nn)	11111101 00101010 n n	Load IY with location nn				
LD (IY+d),n	11111101 00110110 d n	Store n to (IY+d)				
LD (IY+d),r	11111101 01110 r d	Store r to (IY+d)				
LD (nn),A	00110010 n n	Store A to location nn				
LD (nn),dd	11101101 01dd0011 n n	Store register pair to loc'n nn				
LD (nn),HL	00100010 n n	Store HL to location nn				
LD (nn),IX	11011101 00100010 n n	Store IX to location nn				
LD (nn),IY	11111101 00100010 n n	Store IY to location nn				
LD R,A	11101101 01001111	Load R with A				
LD r,r'	01 r r'	Load r with r'				
LD r,n	00 r 110 n	Load r with n				
LD r,(HL)	01 r 110	Load r with (HL)				
LD r,(IX+d)	11011101 01 r 110 d	Load r with (IX+d)				
LD r,(IY+d)	11111101 01 r 110 d	Load r with (IY+d)				
LD SP,HL	11111001	Load SP with HL				
LD SP,IX	11011101 11111001	Load SP with IX				
LD SP,IY	11111101 11111001	Load SP with IY				
LDD	11101101 10101000	Block load, f'ward, no repeat			•	
LDDR	11101101 10111000	Block load, f'ward, repeat			0	
LDI	11101101 10100000	Block load, b'ward, no repeat			•	
LDIR	11101101 10110000	Block load b'ward, repeat			0	
NEG	11101101 01000100	Negate A (two's complement)	•	•	•	•
NOP	00000000	No Operation				
OR r	10110 r	A OR r to A	•	•	•	0
OR n	11110110 n	A OR n to A	•	•	•	0
OR (HL)	10110110	A OR (HL) to A	•	•	•	0

Mnemonic	Format				Description	S	Z	P/V	C
OR (IX+d)	11011101	10110110	d		A OR (IX+d) to A	•	•	•	0
OR (IY+d)	11111101	10110110	d		A OR (IY+d) to A	•	•	•	0
OTDR	11101101	10111011			Block output, b'ward, repeat	•	•	•	
OTIR	11101101	10110011			Block output, f'ward, repeat	•	•	•	
OUT (C),r	11101101	01 r 001			Output r to (C)				
OUT (n),A	11010011	n			Output A to port n				
OUTD	11101101	10101011			Block output, b'ward, no repeat	•	•	•	
OUTI	11101101	10100011			Block output, f'ward, no repeat	•	•	•	
POP IX	11011101	11100001			Pop IX from stack				
POP IY	11111101	11100001			Pop IY from stack				
POP qq	11qq0001				Pop qq from stack				
PUSH IX	11011101	11100101			Push IX onto stack				
PUSH IY	11111101	11100101			Push IY onto stack				
PUSH qq	11qq0101				Push qq onto stack				
RES b,r	11001011	10 b r			Reset bit b or r				
RES b,(HL)	11001011	10 b 110			Reset bit b of (HL)				
RES b,(IX+d)	11011101	11001011	d	10 b 110	Reset bit b of (IX+d)				
RES b,(IY+d)	11111101	11001011	d	10 b 110	Reset bit b of (IY+d)				
RET	11001001				Return from subroutine				
RET cc	11 c 000				Return from subroutine if cc				
RETI	11101101	01001101			Return from interrupt				
RETN	11101101	01000101			Return from non-maskable int				
RL r	11001011	00010 r			Rotate left thru carry r	•	•	•	•
RL (HL)	11001011	00010110			Rotate left thru carry (HL)	•	•	•	•
RL (IX+d)	11011101	11001011	d	00010110	Rotate left thru carry (IX+d)	•	•	•	•
RL (IY+d)	11010101	11001011	d	00000110	Rotate left thru carry (IY+d)	•	•	•	•
RLA	00010111				Rotate A left thru carry				•
RLC r	11001011	00000 r			Rotate left circular r	•	•	•	•
RLC (HL)	11001011	00000110			Rotate left circular (HL)	•	•	•	•
RLC (IX+d)	11011101	11001011	d	00000110	Rotate left circular (IX+d)				
RLC (IY+d)	11111101	11001011	d	00000110	Rotate left circular (IY+d)	•	•	•	•
RLCA	00000111				Rotate left circular A				•
RLD	11101101	01101111			Rotate bcd digit left (HL)	•	•	•	
RR r	11001011	00011 r			Rotate right thru carry r	•	•	•	•
RR (HL)	11001011	00011110			Rotate right thru carry (HL)	•	•	•	•
RR (IX+d)	11011101	11001011	d	00011110	Rotate right thru cy (IX+d)	•	•	•	•
RR (IY+d)	00011110	11001011	d	00011110	Rotate left thru cy (IY+d)	•	•	•	•
RRA	00011111				Rotate A right thru carry				•
RRC r	11001011	00001 r			Rotate r right circular	•	•	•	•
RRC (HL)	11001011	00001110			Rotate (HL) right circular	•	•	•	•
RRC (IX+d)	11011101	11001011	d	00001110	Rotate (IX+d) right circular	•	•	•	•
RRC (IY+d)	11111101	11001011	d	00001110	Rotate (IY+d) right circular	•	•	•	•
RRCA	00001111				Rotate A right circular				•

APPENDIX V Z-80 Instruction Set

Mnemonic	Format	Description	S	Z	P/V	C
RRD	11101101 01100111	Rotate bcd digit right (HL)	•	•	•	
RST p	11 t 110	Restart to location p				
SBC A,r	10011 r	A-r-CY to A	•	•	•	•
SBC A,n	11011110 n	A-n-CY to A	•	•	•	•
SBC A,(HL)	10011110	A-(HL)-CY to A	•	•	•	•
SBC A,(IX+d)	11011101 10011110 d	A-(IX+d)-CY to A	•	•	•	•
SBC A,(IY+d)	11111101 10011110 d	A-(IY+d)-CY to A	•	•	•	•
SBC HL,ss	11101101 01ss0010	HL-ss-CY to HL	•	•	•	•
SCF	00110111	Set carry flag				1
SET b,(HL)	11001011 11 b 110	Set bit b of (HL)				
SET b,(IX+d)	11011101 11001011 d 11 b 110	Set bit b of (IX+d)				
SET b,(IY+d)	11111101 11001011 d 11 b 110	Set bit b of (IY+d)				
SET b,r	11001011 11 b r	Set bit b of r				
SLA r	11001011 00100 r	Shift r left arithmetic	•	•	•	•
SLA (HL)	11001011 00100110	Shift (HL) left arithmetic	•	•	•	•
SLA (IX+d)	11011101 11001011 d 00100110	Shift (IX+d) left arithmetic	•	•	•	•
SLA (IY+d)	11111101 11001011 d 00100110	Shift (IY+d) left arithmetic	•	•	•	•
SRA r	11001011 00101 r	Shift r right arithmetic	•	•	•	•
SRA (HL)	11001011 00101110	Shift (HL) right arithmetic	•	•	•	•
SRA (IX+d)	11011101 11001011 d 00101110	Shift (IX+d) right arithmetic	•	•	•	•
SRA (IY+d)	11111101 11001011 d 00101110	Shift (IY+d) right arithmetic	•	•	•	•
SRL r	11001011 00111 r	Shift r right logical	•	•	•	•
SRL (HL)	11001011 00111110	Shift (HL) right arithmetic	•	•	•	•
SRL (IX+d)	11011101 11001011 d 00111110	Shift (IX+d) right arithmetic	•	•	•	•
SRL (IY+d)	11111101 11001011 d 00111110	Shift (IY+d) right arithmetic	•	•	•	•
SUB r	10010 r	A-r to A	•	•	•	•
SUB n	11010110 n	A-n to A	•	•	•	•
SUB (HL)	10010110	A-(HL) to A	•	•	•	•
SUB (IX+d)	11011101 10010110 d	A-(IX+d) to A	•	•	•	•
SUB (IY+d)	11111101 10010110 d	A-(IY+d) to A	•	•	•	•
XOR r	10101 r	A EXCLUSIVE OR r to A	•	•	•	0
XOR n	11101110 n	A EXCLUSIVE OR n to A	•	•	•	0
XOR (HL)	10101110	A EXCLUSIVE OR (HL) to A	•	•	•	0
XOR (IX+d)	11011101 10101110 d	A EXCLUSIVE OR (IX+d) to A	•	•	•	0
XOR (IY+d)	11111101 10101110 d	A EXCLUSIVE OR (IY+d) to A	•	•	•	0

Key: Instruction Fields: b bit field 0-7
c condition field 0=NZ,1=Z, 2=NC, 3=C
4=PO, 5=PE, 6=P, 7=M
d Indexing displacement +127 to -128
dd register pair: 0=BC, 1=DE, 2=HL, 3=SP
• = affected
• relative jump displacement +127 to -128
0 = reset
n Immediate or address value
1 = set
pp register pair: 0=BC, 1=DE, 2=IX, 3=SP
qq register pair: 0=BC, 1=DE, 2=IY, 3=SP
r register: 0=B, 1=C, 2=D, 3=E, 4=H, 5=L, 7=A
r' register: same as r
ss register pair: 0=BC, 1=DE, 2=HL, 3=SP
t RST field: Location=t*8
Condition Codes:
= unaffected

Appendix VI. Two's Complement Numbers

HEX	BINARY	DECIMAL
7F	01111111	+ 127
7E	01111110	+ 126
7D	01111101	+ 125
7C	01111100	+ 124
7B	01111011	+ 123
7A	01111010	+ 122
79	01111001	+ 121
78	01111000	+ 120
77	01110111	+ 119
76	01110110	+ 118
75	01110101	+ 117
74	01110100	+ 116
73	01110011	+ 115
72	01110010	+ 114
71	01110001	+ 113
70	01110000	+ 112
6F	01101111	+ 111
6E	01101110	+ 110
6D	01101101	+ 109
6C	01101100	+ 108
6B	01101011	+ 107
6A	01101010	+ 106
69	01101001	+ 105
68	01101000	+ 104
67	01100111	+ 103
66	01100110	+ 102
65	01100101	+ 101
64	01100100	+ 100
63	01100011	+ 99
62	01100010	+ 98
61	01100001	+ 97
60	01100000	+ 96
5F	01011111	+ 95
5E	01011110	+ 94
5D	01011101	+ 93
5C	01011100	+ 92
5B	01011011	+ 91
5A	01011010	+ 90
59	01011001	+ 89
58	01011000	+ 88
57	01010111	+ 87
56	01010110	+ 86
55	01010101	+ 85
54	01010100	+ 84
53	01010011	+ 83
52	01010010	+ 82
51	01010001	+ 81
50	01010000	+ 80
4F	01001111	+ 79
4E	01001110	+ 78
4D	01001101	+ 77
4C	01001100	+ 76
4B	01001011	+ 75

APPENDIX VI Two's Complement Numbers

4A	01001010	+ 74
49	01001001	+ 73
48	01001000	+ 72
47	01000111	+ 71
46	01000110	+ 70
45	01000101	+ 69
44	01000100	+ 68
43	01000011	+ 67
42	01000010	+ 66
41	01000001	+ 65
40	01000000	+ 64
3F	00111111	+ 63
3E	00111110	+ 62
3D	00111101	+ 61
3C	00111100	+ 60
3B	00111011	+ 59
3A	00111010	+ 58
39	00111001	+ 57
38	00111000	+ 56
37	00110111	+ 55
36	00110110	+ 54
35	00110101	+ 53
34	00110100	+ 52
33	00110011	+ 51
32	00110010	+ 50
31	00110001	+ 49
30	00110000	+ 48
2F	00101111	+ 47
2E	00101110	+ 46
2D	00101101	+ 45
2C	00101100	+ 44
2B	00101011	+ 43
2A	00101010	+ 42
29	00101001	+ 41
28	00101000	+ 40
27	00100111	+ 39
26	00100110	+ 38
25	00100101	+ 37
24	00100100	+ 36
23	00100011	+ 35
22	00100010	+ 34
21	00100001	+ 33
20	00100000	+ 32
1F	00011111	+ 31
1E	00011110	+ 30
1D	00011101	+ 29
1C	00011100	+ 28
1B	00011011	+ 27
1A	00011010	+ 26
19	00011001	+ 25
18	00011000	+ 24
17	00010111	+ 23
16	00010110	+ 22
15	00010101	+ 21

14	00010100	+ 20
13	00010011	+ 19
12	00010010	+ 18
11	00010001	+ 17
10	00010000	+ 16
0F	00001111	+ 15
0E	00001110	+ 14
0D	00001101	+ 13
0C	00001100	+ 12
0B	00001011	+ 11
0A	00001010	+ 10
09	00001001	+ 9
08	00001000	+ 8
07	00000111	+ 7
06	00000110	+ 6
05	00000101	+ 5
04	00000100	+ 4
03	00000011	+ 3
02	00000010	+ 2
01	00000001	+ 1
00	00000000	+ 0
FF	11111111	- 1
FE	11111110	- 2
FD	11111101	- 3
FC	11111100	- 4
FB	11111011	- 5
FA	11111010	- 6
F9	11111001	- 7
F8	11111000	- 8
F7	11110111	- 9
F6	11110110	- 10
F5	11110101	- 11
F4	11110100	- 12
F3	11110011	- 13
F2	11110010	- 14
F1	11110001	- 15
F0	11110000	- 16
EF	11101111	- 17
EE	11101110	- 18
ED	11101101	- 19
EC	11101100	- 20
EB	11101011	- 21
EA	11101010	- 22
E9	11101001	- 23
E8	11101000	- 24
E7	11100111	- 25
E6	11100110	- 26
E5	11100101	- 27
E4	11100100	- 28
E3	11100011	- 29
E2	11100010	- 30
E1	11100001	- 31
E0	11100000	- 32
DF	11011111	- 33

APPENDIX VI Two's Complement Numbers

DE	11011110	- 34
DD	11011101	- 35
DC	11011100	- 36
DB	11011011	- 37
DA	11011010	- 38
D9	11011001	- 39
D8	11011000	- 40
D7	11010111	- 41
D6	11010110	- 42
D5	11010101	- 43
D4	11010100	- 44
D3	11010011	- 45
D2	11010010	- 46
D1	11010001	- 47
D0	11010000	- 48
CF	11001111	- 49
CE	11001110	- 50
CD	11001101	- 51
CC	11001100	- 52
CB	11001011	- 53
CA	11001010	- 54
C9	11001001	- 55
C8	11001000	- 56
C7	11000111	- 57
C6	11000110	- 58
C5	11000101	- 59
C4	11000100	- 60
C3	11000011	- 61
C2	11000010	- 62
C1	11000001	- 63
C0	11000000	- 64
BF	10111111	- 65
BE	10111110	- 66
BD	10111101	- 67
BC	10111100	- 68
BB	10111011	- 69
BA	10111010	- 70
B9	10111001	- 71
B8	10111000	- 72
B7	10110111	- 73
B6	10110110	- 74
B5	10110101	- 75
B4	10110100	- 76
B3	10110011	- 77
B2	10110010	- 78
B1	10110001	- 79
B0	10110000	- 80
AF	10101111	- 81
AE	10101110	- 82
AD	10101101	- 83
AC	10101100	- 84
AB	10101011	- 85
AA	10101010	- 86
A9	10101001	- 87

Two's Complement Numbers APPENDIX VI

A8	10101000	- 88
A7	10100111	- 89
A6	10100110	- 90
A5	10100101	- 91
A4	10100100	- 92
A3	10100011	- 93
A2	10100010	- 94
A1	10100001	- 95
A0	10100000	- 96
9F	10011111	- 97
9E	10011110	- 98
9D	10011101	- 99
9C	10011100	- 100
9B	10011011	- 101
9A	10011010	- 102
99	10011001	- 103
98	10011000	- 104
97	10010111	- 105
96	10010110	- 106
95	10010101	- 107
94	10010100	- 108
93	10010011	- 109
92	10010010	- 110
91	10010001	- 111
90	10010000	- 112
8F	10001111	- 113
8E	10001110	- 114
8D	10001101	- 115
8C	10001100	- 116
8B	10001011	- 117
8A	10001010	- 118
89	10001001	- 119
88	10001000	- 120
87	10000111	- 121
86	10000110	- 122
85	10000101	- 123
84	10000100	- 124
83	10000011	- 125
82	10000010	- 126
81	10000001	- 127
80	10000000	- 128



.....

Appendix VII. Printable ASCII Codes

HEX	DECIMAL	ASCII
20	32	(space)
21	33	!
22	34	"
23	35	#
24	36	\$
25	37	%
26	38	&
27	39	'
28	40	(
29	41)
2A	42	*
2B	43	+
2C	44	,
2D	45	-
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S

APPENDIX VII Printable ASCII Codes

54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93]
5E	94	^
5F	95	_
60	96	@
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	
7D	125	}
7E	126	~
7F	127	





RADIO SHACK, A DIVISION OF TANDY CORPORATION

U.S.A.: FORT WORTH, TEXAS 76102
CANADA: BARRIE, ONTARIO L4M 4W5

TANDY CORPORATION

AUSTRALIA

91 KURRAJONG ROAD
MOUNT DRUITT, N.S.W. 2770

BELGIUM

PARC INDUSTRIEL DE NANINNE
5140 NANINNE

U. K.

BILSTON ROAD WEDNESBURY
WEST MIDLANDS WS10 7JN